

# KBLAS: An Optimized Library for Dense Matrix-Vector Multiplication on GPU Accelerators

Ahmad Abdelfattah, Extreme Computing Research Center, KAUST

David Keyes, Extreme Computing Research Center, KAUST

Hatem Ltaief, Extreme Computing Research Center, KAUST

KBLAS is a new open source high performance library that provides optimized kernels for a subset of Level 2 BLAS functionalities on CUDA-enabled GPUs. Since performance of dense matrix-vector multiplication is hindered by the overhead of memory accesses, a double-buffering optimization technique is employed to overlap data motion with computation. After identifying a proper set of tuning parameters, KBLAS is able to efficiently run on various GPU architectures across different generations, avoiding the time-consuming step of code rewriting, while still being compliant with the standard BLAS API. Another advanced optimization technique allows to ensure coalesced memory access when dealing with submatrices, especially in the context of high level dense linear algebra algorithms. All four precisions KBLAS kernels have been leveraged to multi-GPUs environment, which requires the introduction of new APIs to ease users' experiences on these challenging systems. The KBLAS performance outperforms existing state-of-the-art implementations on all matrix sizes, achieves asymptotically up to 50% and 60% speedup on single GPU and multi-GPUs systems, respectively, and validates our performance model. A subset of KBLAS high performance kernels has been integrated into NVIDIA's standard BLAS implementation (cuBLAS) for larger dissemination, starting version 6.0.

Categories and Subject Descriptors: G.4 [Mathematical Software]: Performance

General Terms: Design, Algorithms

Additional Key Words and Phrases: Basic Linear Algebra Subroutines, Memory-Bound Kernels, GPU Accelerators, CUDA Optimizations

## ACM Reference Format:

Abdelfattah, A., Keyes, D., and Ltaief, H. 2014. KBLAS: An Optimized Library for Dense Matrix-Vector Multiplication on GPU Accelerators. *ACM Trans. Math. Softw.* V, N, Article A (January YYYY), 30 pages. DOI = 10.1145/0000000.0000000 <http://doi.acm.org/10.1145/0000000.0000000>

## 1. INTRODUCTION

The increasing compute power of modern hardware accelerators, such as GPUs, has drawn interest for general purpose scientific computing, especially compute-intensive workloads, which expose data parallelism at the forefront. Dense Linear Algebra (DLA) is one area where GPUs can achieve orders of magnitude better performance than traditional multi-core architectures. This is because most DLA algorithms, as implemented in the standard LAPACK library [Anderson et al. 1999], are embarrassingly parallel with regular memory accesses and essentially rely on Level 3 BLAS [BLAS] (i.e., matrix-matrix multiplication) for high performance.

Author's addresses: A. Abdelfattah (Ahmad.Ahmad@kaust.edu.sa), D. Keyes (David.Keyes@kaust.edu.sa) and H. Ltaief (Hatem.Ltaief@kaust.edu.sa), Extreme Computing Research Center, King Abdullah University of Science and Technology, Thuwal, Saudi Arabia,

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or [permissions@acm.org](mailto:permissions@acm.org).

© YYYY ACM 0098-3500/YYYY/01-ARTA \$10.00

DOI 10.1145/0000000.0000000 <http://doi.acm.org/10.1145/0000000.0000000>

Matrix-Vector Multiplication (MVM) kernels are also widely used in DLA algorithms. In fact, these Level 2 BLAS operations are the building blocks of the panel factorization phase for one-sided and two-sided transformations, while solving linear systems of equations and eigenvalue problems or singular value decomposition, respectively. However, such kernels have low floating point operations per byte ratio (flops/byte), and so are bounded by the sustained memory bandwidth of the hardware. As part of the critical path, MVM kernels usually represent serious performance bottlenecks in the aforementioned algorithms. In fact, current state-of-the-art MVM implementations on GPUs are capable of extracting only a small percentage of the bus bandwidth peak. Therefore, optimization techniques should maximize the memory bus utilization, in order to push MVM's performance close to the STREAM benchmark [McCalpin 1995; 2007], which represents a tight upper-bound for applications performance hindered by the bus bandwidth.

This paper introduces KBLAS<sup>1</sup>, an optimized library for dense MVM kernels on GPUs. The KBLAS library supports all four standard precisions. It can also run on shared-memory compute nodes equipped with multiple GPUs. For easy integration, the single GPU kernels from KBLAS are fully compliant with the standard BLAS interface. KBLAS also provides new interfaces for advanced users to ensure coalesced memory access, when dealing with operations on submatrices. Moreover, new APIs for multi-GPUs systems are proposed to facilitate user code developments, thanks to a transparent memory management. The authors build over previous work [Abdelfattah et al. 2013a][Abdelfattah et al. 2013b] and introduce more functionalities and performance tuning knobs to maintain decent throughput across previous and current GPUs hardware generations, without code rewriting. The KBLAS performance outperforms existing state-of-the-art open-source and commercial implementations (i.e., NVIDIA's standard BLAS implementation cuBLAS [NVIDIA 2014b], MAGMABLAS [MAGMA 2009] and CULA [Humphrey et al. 2010]) on all matrix sizes. KBLAS achieves asymptotically up to 50% and 60% speedup against the best implementations on single GPU and multi-GPUs systems, respectively. Most of KBLAS kernels run within at least 80% of the sustained peak performance determined by our performance model, which is based on experimentally measuring the memory bandwidth using STREAM benchmark. The paper also shows up to 20% improvement of high performance DLA libraries after KBLAS integration. Last, but not least, a subset of KBLAS high performance kernels has been integrated into cuBLAS for larger dissemination, starting version 6.0<sup>2</sup>.

The rest of the paper is organized as follows. Section 2 presents related work. Section 3 highlights our contributions. Section 4 gives a general overview of GPU architectures. Section 5 describes the KBLAS framework and presents its different features and functionalities. The implementation details of the high performance MVM kernels are given in Section 6. The performance model to support our empirical data is introduced in Section 7. Section 8 shows KBLAS performance results on various systems and compare against the state-of-the-art commercial and open-source high performance MVM implementations. Section 9 identifies critical parameters necessary to tune the KBLAS kernels on different GPU architectures. Section 10 illustrates the performance impact after integrating KBLAS into DLA libraries and we conclude in Section 11.

<sup>1</sup>KBLAS: KAUST Basic Linear Algebra Subprograms. Available at <http://cec.kaust.edu.sa/Pages/kblas.aspx>

<sup>2</sup><http://docs.nvidia.com/cuda/cublas/#appendix-acknowledgements>

## 2. RELATED WORK

DLA solves problems that are usually regular and well structured. Such problems are suitable for acceleration using GPUs. In fact, the GPU accelerated DLA literature is rich in research efforts that shows orders of magnitude performance gain against multi-core architectures.

For example, standard BLAS operations are provided by vendors [NVIDIA 2014b], while researchers keep providing even more optimized BLAS and incidentally LAPACK routines. The early work presented by [Volkov and Demmel 2008] envisions GPU architectures as multi-threaded multicore vector units, capable of achieving high performance dense linear algebra routines, including matrix multiplication and matrix factorizations (QR, LU and Cholesky). The developers of MAGMA [MAGMA 2009] presented a set of optimization techniques to accelerate BLAS operations on GPUs [Nath et al. 2010b; 2011].

Level 3 BLAS operations, especially matrix multiplication are targets of many research efforts that aim to run these operations as close as possible to the theoretical peak performance of the GPU [Volkov and Demmel 2008; Nath et al. 2010a; Tan et al. 2011]. The latter work uses optimization techniques like register and shared memory blocking. In addition, it proposes an optimal implementation using a pseudo-assembly language (Parallel Thread Execution), which runs very close to the peak performance of the GPU.

Level 2 BLAS are more challenging to optimize due to the lack of data reuse to compensate for the data transfer overhead. There are fewer research initiatives in this area. Part of the MAGMA BLAS library is an optimized symmetric MVM kernel [Nath et al. 2011]. This kernel takes advantage of the symmetry, unlike what was provided by cuBLAS at that time. It adopts a *recursive blocking* technique to handle block sizes that do not fit as a whole in shared memory and a *pointer redirecting* technique that prevents memory access violations when the matrix dimension is not divisible by the block size. The kernel uses an extra workspace in the GPU global memory to perform a final reduction. Although rich in synchronization overhead, this implementation still runs more than twice as fast as the cuBLAS kernel.

NVIDIA cuBLAS 5.5 currently provides an optimized kernel for symmetric MVM, which takes advantage of the symmetry. There is also another implementation for the same kernel provided by the commercial library CULA [Humphrey et al. 2010]. The work presented in this paper compares our performance implementations against similar kernels from the aforementioned software libraries i.e., NVIDIA's cuBLAS, MAGMA BLAS, and CULA.

## 3. CONTRIBUTIONS

Our contributions in this paper can be summarized as follows:

- The performance of our previous SYMV/HEMV [Abdelfattah et al. 2013a] and GEMV [Abdelfattah et al. 2013b] kernel implementations have been improved on single GPUs and across different generations, thanks to the identification of tunable critical performance parameters.
- We introduce new interfaces for GEMV and SYMV/HEMV kernels, when operating on submatrices. These new kernel implementations remove the resulting non-coalesced memory accesses due to matrix offset and maintain the original performance improvement, as seen on regular matrices.
- We propose new GEMV and SYMV/HEMV kernels for shared-memory systems equipped with multi-GPUs. These kernels operate on matrices distributed across multi-GPUs. The new interfaces abstract the hardware complexity from end users

and transparently handle the memory management between the host and the device.

- All developed kernels have been released into a single open-source library named KBLAS, available for download under the modified BSD license.

#### 4. OVERVIEW OF GPUS

This section provides an overview of a modern GPU architecture and one of its programming models.

##### 4.1. Hardware Architecture

GPUs represent an example of a many-core architecture. They are built to deliver levels of processing throughput beyond the capability of traditional multi-core architectures, which adopt architectural optimizations to minimize execution latency rather than throughput. A GPU thread executes slower than a CPU thread, but the GPU compensates for this by being able to execute orders of magnitude more concurrent threads than what a CPU can execute. Therefore, GPUs usually outperform CPUs in executing kernels that include large amounts of parallelism.

From an architectural point of view, GPUs do not have complex hardware components that usually achieve fast execution of a single instruction stream. For example, large caches, advanced memory prefetchers, and deep instruction pipelines are not found in today's GPU architectures. Moreover, multiple threads (as of today, typically 32) share the same instruction stream, in order to amortize the complexity of the fetch and decode unit. This group of thread is called a *warp*. The execution model of a GPU is a variant of the Single Instruction Multiple Data (SIMD) model, called the Single Instruction Multiple Threads (SIMT). Several threads execute the same scalar instruction on multiple operands. This is in contrast to the explicit vector instructions executed on long registers, which is the case in modern CPUs and Intel's Many Integrated Cores (MIC) architecture.

A schematic diagram of a typical GPU architecture is shown in Figure 1. As an example, we will mention NVIDIA's Kepler architecture [NVIDIA 2012], which is the GPU used to show the performance results in this paper. The Kepler architecture is similar in many ways to its predecessor, the Fermi GPU [NVIDIA 2009]. It mainly consists of Streaming Multiprocessors (SMs). Each SM consists of a number of simple cores (CUDA cores) capable of doing both integer and floating point operations. Each SM in a Kepler GPU has 192 CUDA cores, equipped with 64 double precision units. There is also a number of special function units (SFUs) that can compute nonlinear functions on 32 bit floating point numbers. The memory hierarchy on a Kepler GPU is simple. Each SM has its own L1 cache/shared memory module, register file, and constant cache. All SMs share an L2 cache and the global DRAM. Texture memory is also globally accessible to SMs.

One of the hardware features used by KBLAS is atomic operations. Different threads can do certain operations atomically on shared and global memories. Atomic operations are much faster on Kepler than on Fermi. As we will point out later, atomic operations are used in almost every kernel in KBLAS.

##### 4.2. Programming Models

A GPU programming model tends to leverage the GPU capabilities in general purpose computing. One of the early efforts in this area was Brook for GPUs [Buck et al. 2004]. GPUs now have many programming models, compilers, and runtimes, such as CUDA, OpenCL, and OpenACC. We focus on the CUDA programming model.

NVIDIA GPUs natively support the CUDA programming model (or simply CUDA), which is used for the development of KBLAS. CUDA provides extensions to widely

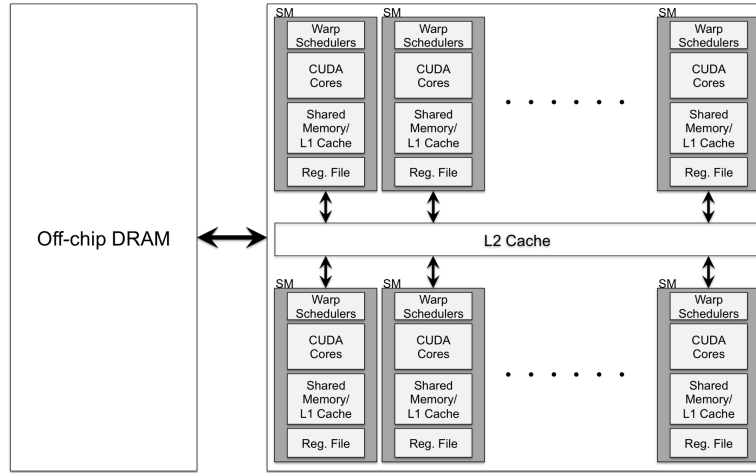


Fig. 1: A modern GPU architecture

used programming languages like C, C++, Fortran, and Java. These extensions enable the development of GPU codes within a CPU code in one context. The main entry of execution for a GPU code is called a *CUDA kernel*. A CUDA kernel is launched by the CPU, or by another CUDA kernel (a feature only in Kepler GPUs, called *Dynamic Parallelism*). All kernel launches on the GPU are asynchronous, as the control returns immediately to the host thread. It is the programmer's responsibility to ensure the completion of any CUDA kernel through the CUDA runtime APIs.

A CUDA kernel is organized as a Grid of thread blocks (TBs). Each TB consists of an array of threads. Both the grid and the thread array can have up to three dimensions. The number of TBs is independent from the number of SMs. A TB is executed exactly by one SM, while one SM can execute multiple TBs, if it has the sufficient resources. TBs in the same kernel cannot communicate or share data. Threads within the same TB, however, can share data and synchronize with each other.

Concurrent kernel execution is also possible on one GPU through *CUDA Streams*, which act as queues where independent kernels are submitted to different streams. Managing dependencies and synchronization among concurrent kernels is the responsibility of the programmer. If no stream is specified for the launch, the kernel is submitted to a default stream. Detailed information about the CUDA programming model can be found in the CUDA programming guide [NVIDIA 2014a].

#### 4.3. Performance Considerations

In order to achieve high performance on a GPU, several considerations have to be taken into account. We will focus on considerations that matter most for memory-bound kernels. More details can be found at [Kirk and Hwu 2010] and [NVIDIA 2014a].

**4.3.1. Memory Coalesced Access.** A memory-bound kernel should access the global memory in a coalesced manner, in order to avoid unnecessary memory traffic. Up to 128 bytes can be transferred between an SM and global memory in one transaction. Therefore, Memory reads and writes, executed per warp, should be performed in fully aligned 128 bytes. Otherwise, a warp memory request will be translated into multiple transactions, which penalizes performance. Matrices stored in global memory are usually padded in order to facilitate memory coalesced accesses, as discussed in Section 5.3.

**4.3.2. GPU Occupancy.** Another consideration is the GPU occupancy. In order to run at full memory bandwidth, the number of threads launched within a kernel should be large enough to saturate the memory bus with coalesced memory requests. Kernels of low occupancy often fail to operate at the peak bandwidth, even if memory coalescing is preserved. However, this does not mean launching the maximum possible number of threads, as the kernel might put pressure on other GPU resources, leading to performance drops. For example, kernels that consume too many registers do not usually operate at peak performance, since excessive register pressure leads to register spilling into global memory.

Another dimension of increasing occupancy comes at the TB level. If the input matrix is relatively small, the number of TBs launched might be too few to fill the GPU resources. It is necessary to ensure that even for small problem sizes, enough TBs are launched. For example, through atomic operations, several TBs can collaboratively work on the same output. Although there will be an overhead due to the increase in atomic operations, the overall performance is often better than launching one TB for the same problem.

**4.3.3. Data Prefetching.** In order to hide the latency of doing FLOPs, a kernel should appear as if it is only performing memory operations. The latency of any computation should be hidden by a memory operation. This can be done through prefetching data while a useful computation is taking place.

**4.3.4. Performance Oscillation.** In some cases, it is important to consider the number of TBs launched with respect to the number of resident SMs on the GPU. Let's assume the simple case when all TBs have a balanced workload. We denote by  $TB_R$  the number of remaining TBs after the partial kernel execution, where all GPU SMs are fully occupied. Typically  $TB_R = \#TBs \bmod \#SMs$ . Performance is penalized if it is relatively low ( $TB_R \ll \#SMs$ ), since the GPU will encounter a duration of low utilization while executing these remaining TBs. As a result, the performance drops for problem sizes that result in low  $TB_R$ , leading to a near-periodic oscillatory behavior in performance.

There are two ways to resolve this problem:

- (1) Detect problem sizes with low  $TB_R$  values, and reconfigure the kernel by adjusting the number of launched TBs to ensure that the value of  $TB_R$  is as close as possible to the number of SMs. This approach requires kernel reconfiguration or even developing a dedicated kernel.
- (2) If the GPU is kept at full utilization for most of the kernel execution time, the impact of the low utilization execution round will be negligible. For example, a kernel launched with 1001 TBs (with balanced workloads) on a 10 SM GPU will result in 100 full execution rounds and 1 round of low utilization. The GPU will be at full utilization for more than 99% of the execution time, and the performance is maintained. This example is valid assuming that an SM executes exactly one TB at a time. In general, a large number of TBs with balanced workloads will ensure that the GPU is fully busy for most of the kernel execution time, and thus performance will be smooth across a wide spectrum of the problem size.

## 5. THE KBLAS LIBRARY

This section highlights all the functionalities supported by KBLAS, as well as the data layout required by its routines.

### 5.1. Main Structure

KBLAS consists mainly of CUDA source codes and testing codes written in C. The CUDA source codes can be categorized into two groups:

- (1) *Kernel Templates*: These are CUDA header files, written in CUDA C/C++ extensions. These files contain CUDA kernels abstracted using C++ templates. The precision as well as the tuning parameters are all template parameters and are never specified in this kind of files. Basic operations like initialization, addition, multiplication, ... etc are all abstracted.
- (2) *Kernel Instances*: These files instantiate, at compile time, a certain precision from the kernel templates. They also specify the tuning parameters with which the instance would be created. These files do not include any CUDA kernel. They provide the APIs that should be used by the user.

The above categorization has several advantages. It maintains one source file per kernel, with the ability to create as many instances as possible. The fact that the tuning parameters are in the instance files makes it straightforward to tune a KBLAS kernel, by simply changing the tuning parameters in the instance file. That is, the core kernels in the template files are not touched. In addition, the tuning parameters are all known at the compile time, which makes it easy for the compiler to optimize the code wherever possible, especially loop unrolling. Finally, implementing further optimizations for a certain kernel means only changing its corresponding template file without touching the instance files.

## 5.2. Routines and APIs

KBLAS implements the standard BLAS operation:

$$y = \alpha Ax + \beta y, \quad (1)$$

where  $y$  is the vector to be updated (input/output),  $x$  is an input vector,  $A$  is an input general or Hermitian matrix, and  $\alpha$  and  $\beta$  are input scalars. A general matrix is processed using the KBLAS implementation of the General Matrix Vector Multiplication (GEMV) kernel. The Hermitian case is processed using either the Symmetric Matrix Vector Multiplication (SYMV), if the matrix is real, or its Hermitian version (HEMV), if the matrix is complex. KBLAS provides these kernels in single and multiple GPUs that exist on the same node. It also provides sophisticated new interfaces for the single GPU routines that can perform better for multiplication by a submatrix. Table I shows all the routines provided by KBLAS. The four supported precisions are represented by the letters s, d, c, and z, which represent the single, double, complex, and double complex precisions respectively. In any variant of the GEMV routine,  $x \in \{s, d, c, z\}$ . For the SYMV routines,  $x \in \{s, d\}$ , while for the HEMV case  $x \in \{c, z\}$ .

Acronym	Matrix Type	API	single GPU	Multi-GPU	Standard Interface
GEMV	General	kblas_xgemv	yes	no	yes
SYMV	Symmetric	kblas_xsymv	yes	no	yes
HEMV	Hermitian	kblas_xhemv	yes	no	yes
GEMV-OFFSET	General	kblas_xgemv_offset	yes	no	no
SYMV-OFFSET	Symmetric	kblas_xsymv_offset	yes	no	no
HEMV-OFFSET	Hermitian	kblas_xhemv_offset	yes	no	no
GEMV-MGPU	General	kblas_xgemv_mgpu	no	yes	no
SYMV-MGPU	Symmetric	kblas_xsymv_mgpu	no	yes	no
HEMV-MGPU	Hermitian	kblas_xhemv_mgpu	no	yes	no

Table I: KBLAS Supported Routines

There are two groups of KBLAS routines that do not have a standard interface in Table I. The first one provides routines that work on single GPU, and perform better than similar standard routines in the case of multiplication by a submatrix. The second group of routines works on multi-GPUs that belong to the same node. According to our knowledge, there is no standard interface multi-GPU BLAS. KBLAS uses an interface similar to the one proposed in [Yamazaki et al. 2013].

Every routine in KBLAS has an asynchronous version, which has the same name appended by the "\_async" suffix. For example, `kblas_sgemv` has a variant called `kblas_sgemv_async`. Default routines are synchronous in the sense that they are launched within the default stream on the GPU. Kernels that are submitted within this stream do not overlap execution with other kernels in different streams. The asynchronous version gives the user the ability to launch KBLAS kernels into a user-defined stream, potentially to overlap its execution with other tasks on the GPU. The asynchronous versions have an extra input argument in order to specify the launch stream.

KBLAS also provides few APIs that facilitate using the multi-GPU routines. It provides functions that allocate the necessary memory space on each GPU in order to launch the multi-GPU routine. It also provides functions that offload the matrix to the GPUs involved in computation, based on the 1D cyclic block column layout. Since the size of the block column is required, KBLAS exposes such values through another set of functions. More details about KBLAS routines and APIs can be found in the KBLAS User's Guide.

### 5.3. Data Layout

**5.3.1. Single GPU Routines.** KBLAS supports the BLAS standard column major format for storing matrices, which is the interface used by the standard BLAS and LAPACK libraries. For single GPU routines, a matrix is described using its dimensions, a pointer to the first element in the matrix, and the leading dimension (LD), which gives the distance (in elements) between two adjacent elements in the same row. Unless the matrix is padded, the leading dimension is equal to the number of rows of the matrix. Figure 2 shows the column major storage with both unpadded and padded leading dimensions. A padded LD is used to preserve memory coalesced access.

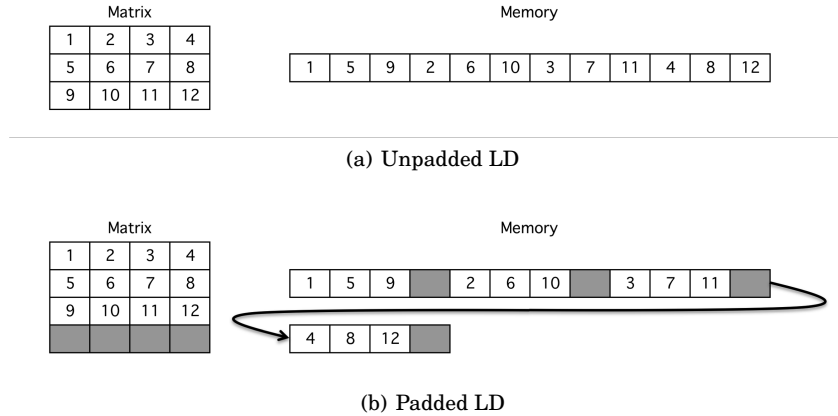


Fig. 2: Column major layout for single GPU kernels



**5.3.2. Multi-GPU Routines.** If the matrix is processed using a multi-GPU routine, it will be distributed among the GPUs involved in computation. The data layout has to be suitable for higher-level algorithms utilizing matrix vector multiplications. We pay attention to a recent work, which is part of the MAGMA library, which provides a multi-GPU tridiagonal reduction for dense symmetric matrices [Yamazaki et al. 2013]. In the MAGMA implementation, the matrix is distributed among GPUs in a 1D cyclic manner, by block columns. KBLAS uses the same layout for its multi-GPU routines, since it seeks to accelerate such reduction algorithms in open source libraries. KBLAS provides auxiliary routines for the allocation and distribution of matrices over multi-GPU for the aforementioned layout. Figure 3 shows an example for a dense symmetric matrix distributed over four GPUs. Each cell (block) represents a square submatrix. The black cells marked with ‘D’ are the diagonal blocks.

By default, each GPU keeps a local copy of the original vectors  $x$  and  $y$ . However, in some KBLAS kernels,  $y$  is distributed, in segments, in a 1D cyclic manner. KBLAS provides routines for the distribution of the vector among GPUs. Output vectors, upon kernel termination, are sent back to the CPU, where a final reduction is performed to obtain the result.

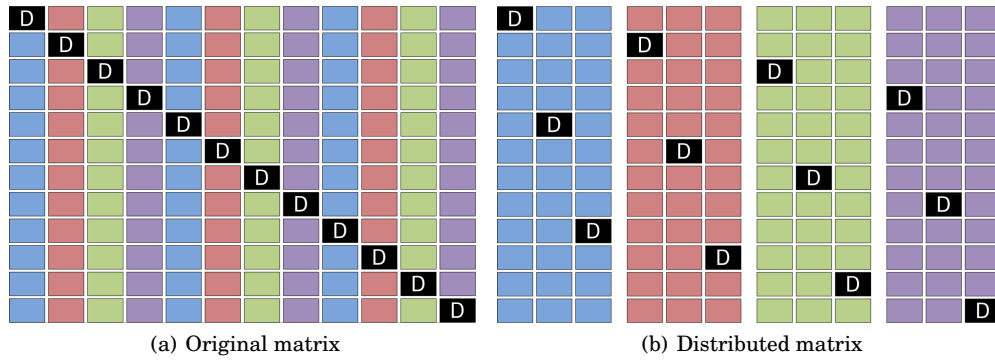


Fig. 3: Matrix layout for multi-GPU kernels

## 6. HIGH PERFORMANCE KERNEL IMPLEMENTATIONS

This section presents a detailed description of the implementation of KBLAS kernels. We will begin with single GPU kernels and leverage the design idea to the multi-GPU kernels.

### 6.1. General Outlines

We focus on describing the processing pattern of input matrices, since this is the dominant part when compared with reading or writing input and output vectors. Input matrices are processed in square blocks. The block size  $nb$  is a tuning parameter.

KBLAS usually launches more than one kernel per a BLAS operation. Sections 6.2 and 6.3 discuss the design of kernels that dominate the execution time of the BLAS operations. The dominant kernels are:

- (1) The general non-transposed MV kernel computing the product  $\alpha Ax$ .
- (2) The general transposed MV kernel computing the product  $\alpha A^T x$ .
- (3) The symmetric/Hermitian MV kernel computing the product  $\alpha \hat{A} x$ , where  $\hat{A}$  refers to the off diagonal blocks of  $A$ .

These kernels have a common processing strategy, and are designed in almost the same manner. Other kernels will be discussed separately in Sections 6.4 and 6.5.

## 6.2. Grid Design

In general, the grid of any KBLAS kernel is organized as a 2D array of TBs. The size of the grid is  $(\bar{X}, \bar{Y})$ . While  $\bar{Y}$  is a tuning parameter that the user can pick regardless of the problem size,  $\bar{X}$  is decided based on the problem size and  $nb$ . Given a matrix  $A_{m \times n}$ ,  $\bar{X}$  is given by,

$$\bar{X} = \begin{cases} \lceil \frac{m}{nb} \rceil & ; \text{A is not transposed} \\ \lceil \frac{n}{nb} \rceil & ; \text{A is transposed} \end{cases} \quad (2)$$

The value of  $\bar{Y}$  determines the number of TBs working collaboratively (through atomic operations) on the same part of the output vector. From now on, we will consider square matrices, that is,  $m = n = d$ .

Each TB is distinguish by its  $(\bar{x}, \bar{y})$  coordinates. The  $\bar{x}$  coordinate indicates the block row or the block column on which the TB will work. Eventually, TBs having different values of the  $\bar{x}$  write to different parts of the output vector. TBs with different  $\bar{y}$  coordinate and share the same  $\bar{x}$  coordinate will write to the same segment in the output vector using atomic operations.

All TBs traverse the input matrix, in blocks, either horizontally or vertically. The movement direction depends on the operation to be performed. In the case of a GEMV operation, the movement is decided by the input character that specifies whether the matrix is transposed. If it is, then TBs will move vertically. Otherwise, the movement is horizontal. Figure 4 shows both possibilities. In either case, TBs are programmed

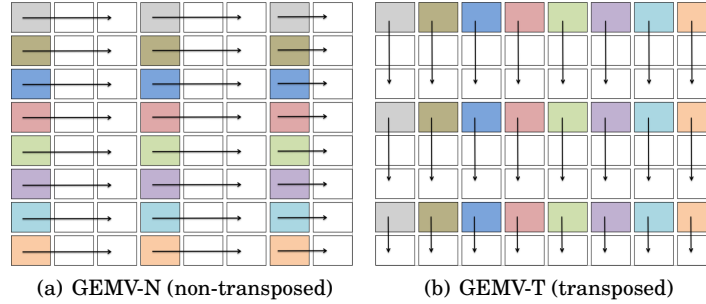


Fig. 4: Movement of TBs in a GEMV operation. TBs with the same color share the same value of  $\bar{x}$ .

to traverse the entire matrix. At the beginning of execution, each TB must decide its starting point (first block) and its workload (number of blocks to be processed by this TB). TBs sharing the same  $\bar{x}$  values will collaboratively process an entire block row or an entire block column of the matrix. The total workload  $W$  for these TBs is given by

$$W = \left\lceil \frac{d}{nb} \right\rceil \quad (3)$$

Then each TB computes its workload share  $w$  and its starting point  $s$  as follows

$$w = \frac{d}{\bar{Y}} + \begin{cases} 0; & \bar{y} \geq d \bmod \bar{Y} \\ 1; & \bar{y} < d \bmod \bar{Y} \end{cases} \quad (4)$$

$$s = \bar{y} \left\lfloor \frac{W}{\bar{Y}} \right\rfloor + \min(\bar{y}, W \bmod \bar{Y}) \quad (5)$$

Equation 4 ensures a minimum load imbalance among TBs. Equation 5 allows each TB to process adjacent blocks in the matrix. According to the input matrix size and grid configuration, some TBs might have their  $w$  value equal to zero. In such a case, these TBs terminate immediately, writing nothing to the output vector.

If the matrix is symmetric (SYMV/HEMV kernel), then TBs are programmed to process either the upper or the lower triangular part. As shown in Figure 5, diagonal blocks, colored in solid black, are processed separately. This is because they have a different processing pattern from off-diagonal blocks. Off-diagonal blocks are processed in a similar manner to the GEMV kernel, except for the movement being vertical regardless of which triangular part is being processed. The vertical movement ensures contiguous data access for each TB. The total workload  $W$  is not constant across block columns, and is no longer determined by Equation 3. In fact, each block row/block column has a unique value of  $W$  that is given by

$$W = \begin{cases} \left\lceil \frac{d}{nb} \right\rceil - i - 1 & ; \text{lower triangular} \\ i & ; \text{upper triangular} \end{cases}, \quad (6)$$

where  $i$  is the index of a block column, typically  $0, 1, 2, \dots$ , from left to right.

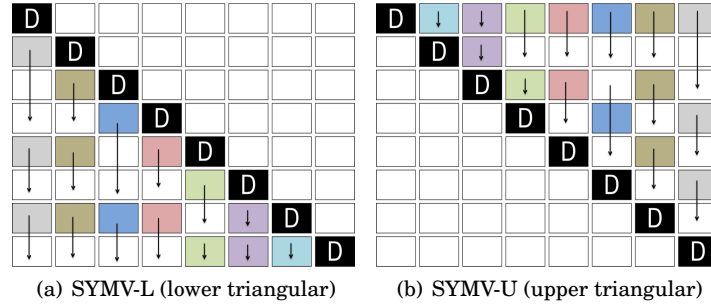


Fig. 5: Movement of TBs in a SYMV/HEMV operation. TBs with the same color share the same value of  $\bar{x}$ .

The same strategy for TB movement applies for the multi-GPU kernels. The movement is horizontal only for the non-transposed GEMV, and vertical otherwise. The different is that the movement will be applied to the local submatrix stored on the GPU, as shown in Figure 3(b), instead of the whole matrix.

### 6.3. Thread Block Design

Each TB is designed as a 2D array of threads. The size of the TB is  $(\bar{P}, \bar{Q})$ , where  $\bar{P}=nb$  and  $\bar{Q}$  is a third tuning parameter. Each thread is distinguished by its  $(\bar{p}, \bar{q})$  coordinates. For a certain value of  $nb$ ,  $\bar{Q}$  controls the total number of threads, the amount of shared memory required for local reductions, and most importantly, the register pressure within a single TB. For simplicity, we will discuss an example of  $\bar{P}=nb=32$  and  $\bar{Q}=4$ , shown in Figure 6.

The design of TBs maximizes memory throughput through double buffers. Each square block of the matrix is processed in two phases, as shown in Figure 6. At each phase a half block of dimension  $\frac{nb}{2} \times nb$  is processed in register buffers. Meanwhile, a new half block is fetched from memory into another set of register buffers. The overlap

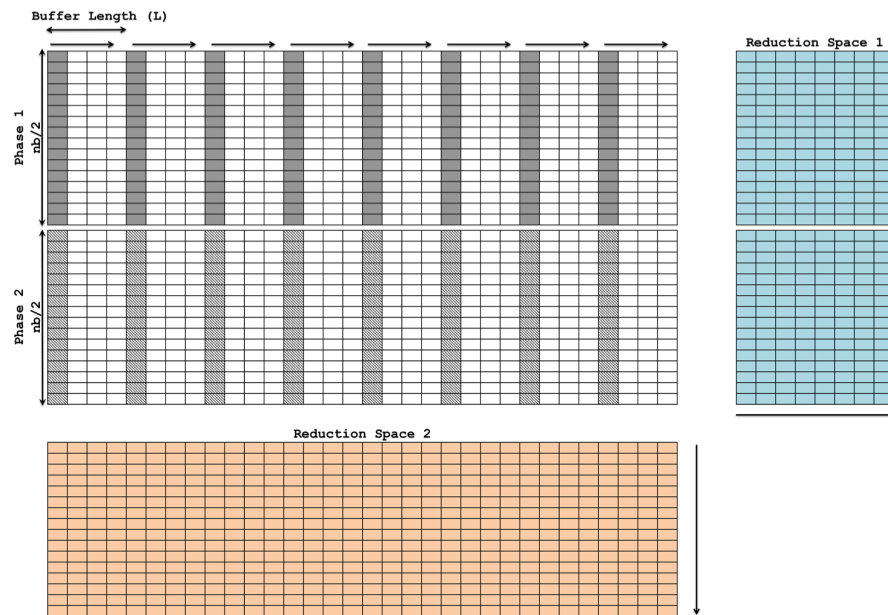


Fig. 6: Thread block design

between computation and memory prefetching spans multiple blocks of the matrix, meaning that processing in phase 2 overlaps phase 1 in the next block. The dark cells in Figure 6 show the original positions of threads in the first half block. The hashed cells indicate the respective positions for these threads in the next half block. Threads are reorganized as a  $\frac{nb}{2} \times 2Q$  thread block ( $16 \times 8$ ). That is, we end up with 8 thread columns, each consisting of 16 threads. A thread column is responsible for processing a  $\frac{nb}{2} \times L$  ( $16 \times 4$ ) rectangle, where  $L$  is the register buffer length required, per thread, per half block.  $L$  is given by

$$L = \frac{nb}{2Q} \quad (7)$$

Each thread keeps two register buffers of length  $L$  to fetch its corresponding elements. For the example discussed in Figure 6, this means thread (0, 0) will be responsible for elements  $\{(0, 0), (0, 1), (0, 2), (0, 3)\}$  as well as elements  $\{(16, 0), (16, 1), (16, 2), (16, 3)\}$ . Upon completion of processing, each thread spills its partial products to a reduction space located in shared memory. A synchronization point is enforced to make sure that all threads have written their partial products. A reduction is done in shared memory before writing the result (using atomic operations) to the global memory.

Thread columns always read the block horizontally, in order to maintain contiguous memory access. However, the number of partial products per thread as well as the frequency of the synchronization and reduction depends on the kernel being executed. All possibilities are summarized using a pseudo code in Figure 7. In the pseudo code, abbreviations *LHB* and *UHB* refer to Lower Half Block and Upper Half Block, respectively. Variables *ures*, *lres*, and *vres*[] refer to register accumulators where each thread keeps its local partial result. The variables *u*[] and *l*[] are register buffers that are used to read elements from *UHB* and *LHB* respectively.

If the kernel is a non-transposed GEMV (Algorithm 1), then only reduction space 1 is needed for the final reduction. Each thread produces 2 partial products, one for each half block. The reduction is performed per row. The total space required for reduction space 1 is given by,

$$\text{reduction space 1} = nb \times (2 \times \bar{Q}) \quad (8)$$

The reduction in this case is needed only once. According to the grid design in Section 6.2, TBs traverse horizontal adjacent blocks, accumulating the partial results of all blocks together. The overhead of synchronization and reduction is, therefore, negligible.

If the kernel is a transposed GEMV (Algorithm 2), then only reduction space 2 is needed for the final reduction. Each thread produces  $L$  partial products, one for each matrix column. The reduction is performed per column. The total space required for reduction is given by,

$$\text{reduction space 2} = \frac{nb}{2} \times nb \quad (9)$$

Like the non-transposed GEMV, the reduction is needed only once, since TBs traverse the matrix vertically, and can accumulate partial results from one block to another.

If the kernel is a SYMV/HEMV (Algorithm 3), exclusive of diagonal blocks, then both reduction spaces are needed. Each off-diagonal block is processed twice: non-transposed and transposed. Each thread will have a total of  $2+L$  partial products. This case combines the two possibilities of the GEMV kernel. However, it differs in the frequency of sync-and-reduce steps. Since TBs always traverse half the matrix vertically, partial results of the transposed computation can be accumulated, but those of non-transposed computation cannot, since they belong to different parts in the output vector. This means that a sync-and-reduce step is required whenever a TB moves from one matrix block to another, in order to write the partial products into global memory. The number of reductions is equal to the number of processed off-diagonal blocks plus an extra reduction performed only once for transposed computation.

The design of TBs does not change going from one GPU to multi-GPUs. The scope of the design is a square block of the matrix, so no major differences in the design exist.

#### 6.4. Scaling with $\beta$ in GEMV kernels

Sections 6.2 and 6.3 show how KBLAS computes the product  $\alpha Ax$ . However, a standard BLAS operation involves scaling  $y$  with  $\beta$ . Although the scaling is a trivial operation, it cannot be performed inside the KBLAS kernel that computes  $\alpha Ax$ . This is because every segment of length  $nb$  in the resulting vector is computed by multiple thread blocks using atomic operations. Since the CUDA programming model handles the executing of TBs transparently, we cannot determine the order of execution for TBs. This means we cannot assign the scaling operation to a particular TB, since it is not necessarily executed first.

The solution is to perform the scaling operation in a separate kernel. The scale and the multiplication kernel are launched in order, so the scale operation must finish before the multiplication is performed. The scale kernel is similar to the standard level-1 BLAS operation SCAL. A KBLAS GEMV operation consists, therefore, of two successive kernels called transparently to the user. As discussed in Section 6.5, the scale kernel is not needed if the matrix is symmetric/Hermitian.

#### 6.5. Diagonal Blocks Processing in SYMV/HEMV kernels

If the matrix is symmetric/Hermitian, then diagonal blocks are different from the off-diagonal blocks, in terms of the processing strategy. The difference is that only one

**ALGORITHM 1: Non-transposed GEMV**


---

**Data:**  $A, x, \alpha$   
**Result:**  $y$   
 Compute  $w$  and  $s$  and navigate accordingly;  
 $ures = lres = 0$  ;  
 $u[] \leftarrow UHB$ ;  
**for**  $j \leftarrow 1$  **to**  $w$  **do**  
    $l[] \leftarrow LHB$ ;  
    $ures \leftarrow ures + UHB \times x$ ;  
   **if**  $j \neq w$  **then**  
     Move right to the next block;  
      $u[] \leftarrow UHB$ ;  
   **end**  
    $lres \leftarrow lres + LHB \times x$ ;  
**end**  
 Write  $ures$  and  $lres$  to SHMEM;  
 Barrier;  
**if**  $\bar{q} = 0$  **then**  
    $r \leftarrow$ reduction in SHMEM;  
   Write  $\alpha \times r$  into  $y$  using atomics;  
**end**

---

**ALGORITHM 2: Transposed GEMV**


---

**Data:**  $A, x, \alpha$   
**Result:**  $y$   
 Compute  $w$  and  $s$  and navigate accordingly;  
 $vres[] = 0$  ;  
 $u[] \leftarrow UHB$ ;  
**for**  $j \leftarrow 1$  **to**  $w$  **do**  
    $l[] \leftarrow LHB$ ;  
    $vres[] \leftarrow vres[] + UHB^T \times x$ ;  
   **if**  $j \neq w$  **then**  
     Move down to the next block;  
      $u[] \leftarrow UHB$ ;  
   **end**  
    $vres[] \leftarrow lres + LHB^T \times x$ ;  
**end**  
 Write  $vres[]$  to SHMEM;  
 Barrier;  
**if**  $\bar{q} = 0$  **then**  
    $r \leftarrow$ reduction in SHMEM;  
   Write  $\alpha \times r$  into  $y$  using atomics;  
**end**

---

**ALGORITHM 3: Upper/Lower SYMV/HEMV**


---

**Data:**  $A, x, \alpha$   
**Result:**  $y$   
 Compute  $w$  and  $s$  and navigate accordingly;  
 $ures = lres = vres[] = 0$  ;  
 $u[] \leftarrow UHB$ ;  
**for**  $j \leftarrow 1$  **to**  $w$  **do**  
    $l[] \leftarrow LHB$ ;  
    $ures \leftarrow ures + UHB \times x$ ;  
    $vres[] \leftarrow vres[] + UHB^T \times x$ ;  
   **if**  $j \neq w$  **then**  
     Move down to the next block;  
      $u[] \leftarrow UHB$ ;  
   **end**  
    $lres \leftarrow lres + LHB \times x$ ;  
    $vres[] \leftarrow vres[] + LHB^T \times x$ ;  
   Barrier;  
   Write  $ures$  and  $lres$  to SHMEM;  
   Barrier;  
   **if**  $\bar{q} = 0$  **then**  
      $r \leftarrow$ reduction in SHMEM;  
     Write  $\alpha \times r$  into  $y$  using atomics;  
   **end**  
**end**  
 Write  $vres[]$  to SHMEM;  
 Barrier;  
**if**  $\bar{q} = 0$  **then**  
    $r \leftarrow$ reduction in SHMEM;  
   Write  $\alpha \times r$  into  $y$  using atomics;  
**end**

---

Fig. 7: Pseudo code of the core kernels in KBLAS

triangular part of a diagonal block should be read from global memory. Moreover, only one non-transposed computation per diagonal block is necessary, in contrast with two computations per a off-diagonal block. KBLAS processes diagonal blocks in a separate kernel.

One TB is launched per diagonal block. The entire block is fetched into shared memory. Then a mirroring step is performed to copy one triangular part to the other. This mirroring step eliminates the triangular part that should not be referenced, meaning that extra reads are done from global memory. After mirroring is done, the block residing in shared memory is multiplied by the corresponding segment in  $x$ . The contribution of this kernel to the total execution time of a SYMV/HEMV operation is negligible, specially if the matrix is large.

The reason behind separation is to simplify the programming, since the two kernels obviously need different amounts of GPU resources. It is also better for the CUDA runtime to optimize the GPU utilization for each kernel individually.

Furthermore, it is possible to fuse the scaling operation with  $\beta$  into this kernel, since exactly one TB is launched per diagonal block. The KBLAS SYMV/HEMV operation does not invoke the SCAL kernel mentioned in Section 6.4.

### 6.6. Multiplication by a Submatrix

In many LAPACK algorithms, matrix-vector multiplication is performed on submatrices. This is a typical situation in matrix reduction techniques. A common performance consideration is to pad the leading dimension so that the matrix can be stored in a fully 128 byte aligned memory space, and so, can be accessed in a coalesced manner. However, if the multiplication is done by a submatrix, coalesced access is not guaranteed and depends on the shifts in rows and columns. For example, Figure 8 shows a matrix stored in a column major format. Each column consists of segments that are stored in aligned 128 bytes. If the entire matrix is to be processed using 8 thread columns, then it is straightforward to write a kernel that maps thread columns to access the matrix in a coalesced manner as shown in Figure 8(a). Now consider the submatrix obtained by skipping one row and one column of the original matrix, as shown in Figure 8(b). Using the same kernel will result in non-coalesced memory access. This is because each memory request by a thread column will translate into 2 memory transactions, leading to an overhead in memory traffic.

KBLAS proposes an additional set of BLAS routines with new interfaces other than the BLAS interfaces. As shown in Figure 8(c), the routine processes the same original matrix, but ignores the top rows and the left most column when computation is performed. This strategy does extra reads, but preserves memory coalesced access. The hashed cells refer to matrix elements that are read but ignored in computation. This is in contrast with Figure 8(b), where the hashed cells correspond to locations outside the matrix boundary that should be avoided by thread columns.

In general, given an  $m \times n$  matrix  $A$ , with a properly padded leading dimension, a multiplication by a submatrix  $\hat{A}$  with the dimensions  $\hat{m} \times \hat{n}$  can be translated into a multiplication by the submatrix  $\bar{A}$  with dimensions  $\bar{m} \times \bar{n}$ , where

$$\bar{m} = \min(m, \left\lceil \frac{\hat{m}}{nb} \right\rceil \times nb) \quad \text{and} \quad \bar{n} = \min(n, \left\lceil \frac{\hat{n}}{nb} \right\rceil \times nb) \quad (10)$$

Equation 10 shows that the dimensions  $\hat{m}$  and  $\hat{n}$  are padded to the nearest values divisible by  $nb$ . This helps minimize the amount of extra global memory reads.

The standard BLAS interface cannot pass information about the input matrix being a part of a bigger one. This is why we propose a new interface to convey these information to the kernel. The user should pass a pointer to the original matrix as well as the

offsets in rows and columns. Since KBLAS relies on perfect memory coalesced access while reading the matrix, these set of new routines are expected to perform better than standard ones when a submatrix is processed.

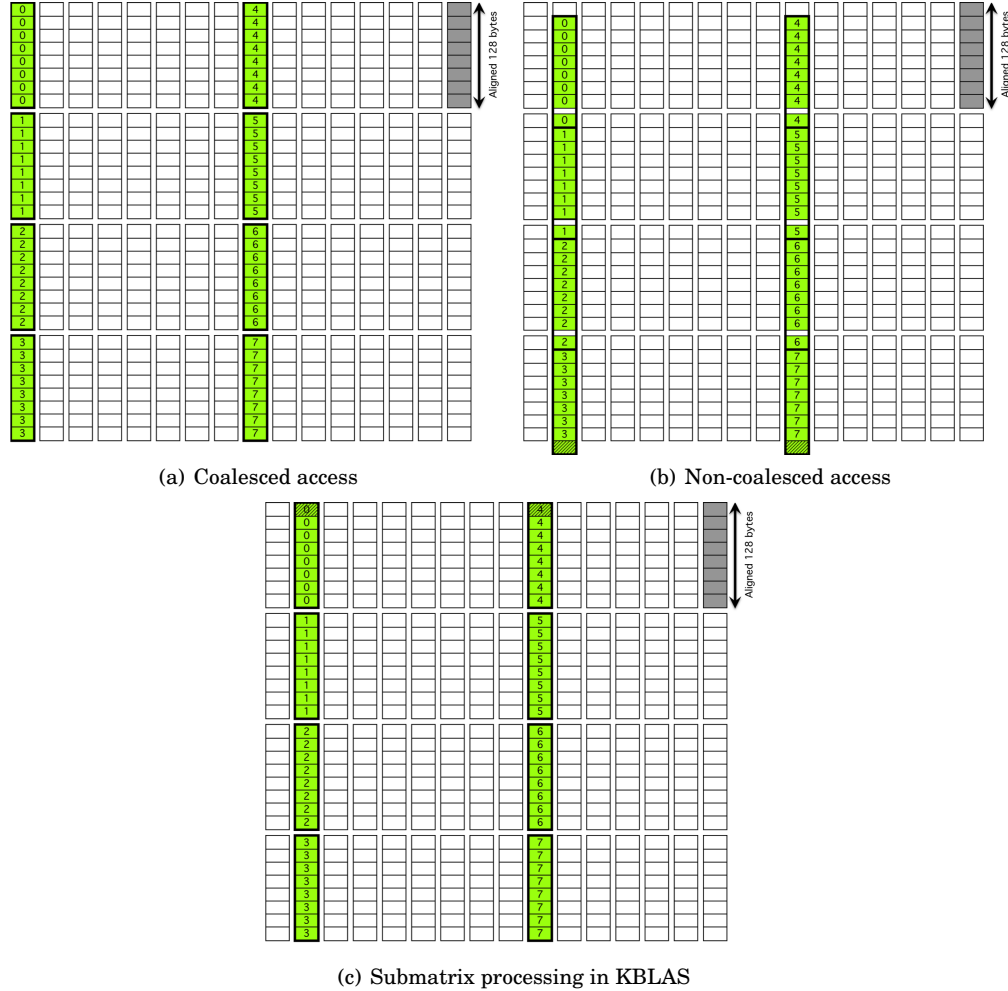


Fig. 8: Non-coalesced memory access due to processing a submatrix

## 7. PERFORMANCE MODEL

We conduct a simple roofline model [Williams et al. 2009] to predict the sustained peak performance of MV kernels. All the analysis in this section applies to a K20c GPU. For a memory-bound kernel, the roofline can be identified using two components:

- (1) The sustained memory bandwidth of the architecture ( $\bar{B}_{max}$ ). This can be experimentally obtained by running sophisticated micro-benchmarks, like the STREAM benchmark [McCalpin 1995; 2007].



ECC	Copy	Scale	Add	Triad
On	148.99	150.64	149.99	150.09
Off	172.44	172.33	175.24	175.24

Table II: STREAM performance in GB/s on a K20c GPU

- (2) The *operational intensity* of the kernel ( $I$ ). This is the ratio of the number of FLOPs executed to the number of bytes read or written to the memory. For some kernels, where data reuse is available, estimating this ratio is not straightforward, and cache hits and misses must be taken into account. However, in our case, no data reuse exists for the input matrix, which is the dominant part regarding both FLOPs and byte counts. The operational intensity of all KBLAS kernels are simple to estimate.

The sustained peak performance for a given kernel ( $R$ ) is, then, given by:

$$R = I \times \bar{B}_{max} \quad (11)$$

The limiting factor for performance is the sustained peak memory bandwidth  $\bar{B}_{max}$ . It should be noted that  $\bar{B}_{max}$  is different from the theoretical peak bandwidth  $B_{max}$ . The latter is theoretically computed from the memory bus width and clock rate. The former is experimentally obtained using a GPU implementation of the STREAM benchmark [McCalpin 1995; 2007]. Considering a K20c GPU, the memory bus width is 320 bits, and the memory clock rate is 2.6 GHz. The theoretical peak bandwidth is given by:

$$B_{max} = 2.6\text{GHz} \times 40 \text{ bytes} \times 2 \text{ (DDR)} = 208 \text{ GB/s} \quad (12)$$

However, in order to get the sustainable memory bandwidth, a micro benchmark needs to run on the GPU and saturate its bus through simple memory operations. A typical STREAM benchmark runs four types of operations: *copy*, *scale*, *add*, and *triad*. Through CUDA kernel implementation for each of these operations, we are able to obtain the sustainable memory bandwidth on a K20c GPU. Results are summarized in Table II. The peak bandwidth on a K20c is 150.64 GB/s (ECC on), and 175.24 GB/s (ECC off). All performance results in this paper are reported with ECC turned off. Therefore,  $\bar{B}_{max}$  is equal to 175.24 GB/s from now on.

The operational intensity ( $I$ ) is simply the number of FLOPs executed divided by the number of bytes transferred between the CUDA cores and the DRAM. We will consider square matrices only for simplicity. An operation  $y_{n \times 1} = \alpha A_{n \times n} x_{n \times 1} + \beta y_{n \times 1}$  involves, per output element in  $y$ , a vector product ( $n$  multiplications +  $n - 1$  additions), two scalar multiplications (for  $\alpha$  and  $\beta$ ), and one addition for the final output. This sums up to a total of  $(n + 2)$  multiplications and  $(n)$  additions per element, and  $(n^2 + 2n)$  multiplications and  $n^2$  additions for the entire operation. This analysis applies to real precisions only. When using a complex precision, a single complex addition maps to 2 additions, and a single complex multiplication maps to 4 multiplications and 2 additions, summing up to 6 FLOPs in total. The total number of real multiplications and real additions in the complex case is, therefore,  $(4n^2 + 4n)$  and  $(4n^2 + 8n)$  respectively.

While the FLOP count is the same for the GEMV and the SYMV/HEMV kernels, the byte count is not. Assuming that  $x$  will *ideally* be read once, and  $y$  is *ideally* read and written once, the total byte count for the GEMV kernel is  $(n^2 + 3n) \times b$ , where  $b$  is the number of bytes used to represent one element in a certain precision. If the matrix is symmetric/Hermitian, either the upper or the lower triangular part is read. This gives a byte count of  $(\frac{n(n+1)}{2} + 3n) \times b$ .

Precision	GEMV	SYMV/HEMV
S	$\frac{2n^2+2n}{4(n^2+3n)} \approx 0.50$	$\frac{2n^2+2n}{4(0.5n^2+3.5n)} \approx 1.00$
D	$\frac{2n^2+2n}{8(n^2+3n)} \approx 0.25$	$\frac{2n^2+2n}{8(0.5n^2+3.5n)} \approx 0.50$
C	$\frac{8n^2+12n}{8(n^2+3n)} \approx 1.00$	$\frac{8n^2+12n}{8(0.5n^2+3.5n)} \approx 2.00$
Z	$\frac{8n^2+12n}{16(n^2+3n)} \approx 0.50$	$\frac{8n^2+12n}{16(0.5n^2+3.5n)} \approx 1.00$

Table III: Operational intensities for KBLAS kernels

Precision	GEMV	SYMV/HEMV
S	87.62	175.24
D	43.81	87.62
C	175.24	338.90
Z	87.62	175.24

Table IV: Estimated sustained peak performances (Gflop/s) for KBLAS kernels on a single K20c GPU (ECC off)

Finally, we can ignore the first order terms in both flop and byte counts, if  $n$  is large enough. Table III summarizes the approximate operational intensities for the GEMV and the SYMV/HEMV kernels in all four precisions.

## 8. PERFORMANCE RESULTS AND ANALYSIS

### 8.1. System Setup

The single GPU experiments are conducted on a system with 16-core Intel Xeon CPU E5-2650 (2.00GHz) and a Tesla K20c GPU (ECC off). The system runs Ubuntu 14.04.1 LTS, CUDA driver version 340.32, and CUDA Toolkit 5.5. The multi-GPU experiments are conducted on a system with a 16 core Intel Xeon CPU E5-2670 (2.60GHz), and equipped with 8 K20c GPUs (ECC off). The system runs CentOS release 6.3, CUDA driver version 331.62, and CUDA Toolkit 5.5. On both systems, we compare the performance of KBLAS against cuBLAS-5.5, MAGMABLAS-1.4.1, and CULA-R17. All results are properly averaged among multiple runs.

### 8.2. Single GPU Performance

Considering a K20c GPU with ECC turned off, Table IV translates the operational intensities listed in Table III into their respective sustained peak performances. We will use these values to estimate how close KBLAS kernels are to their estimated performance bounds.

*8.2.1. Performance of GEMV.* Figure 9 shows, in all precisions, the performance of the GEMV kernel. KBLAS, along with other libraries, is capable of asymptotically score the sustained peak performance, as determined by Table IV, according to the performance model in Section 7. This is the case in all precisions, except for the double complex precision, where KBLAS-ZGEMV has an performance improvement up to 40% against the best competitor. We also note that in some cases, the performance scored by MAGMABLAS or CULA are identical to cuBLAS, which suggests that sometimes the vendor's implementation is invoked internally. We also observe that KBLAS has a is able to maintain its smooth performance, unlike other implementations that suffer from performance oscillations.

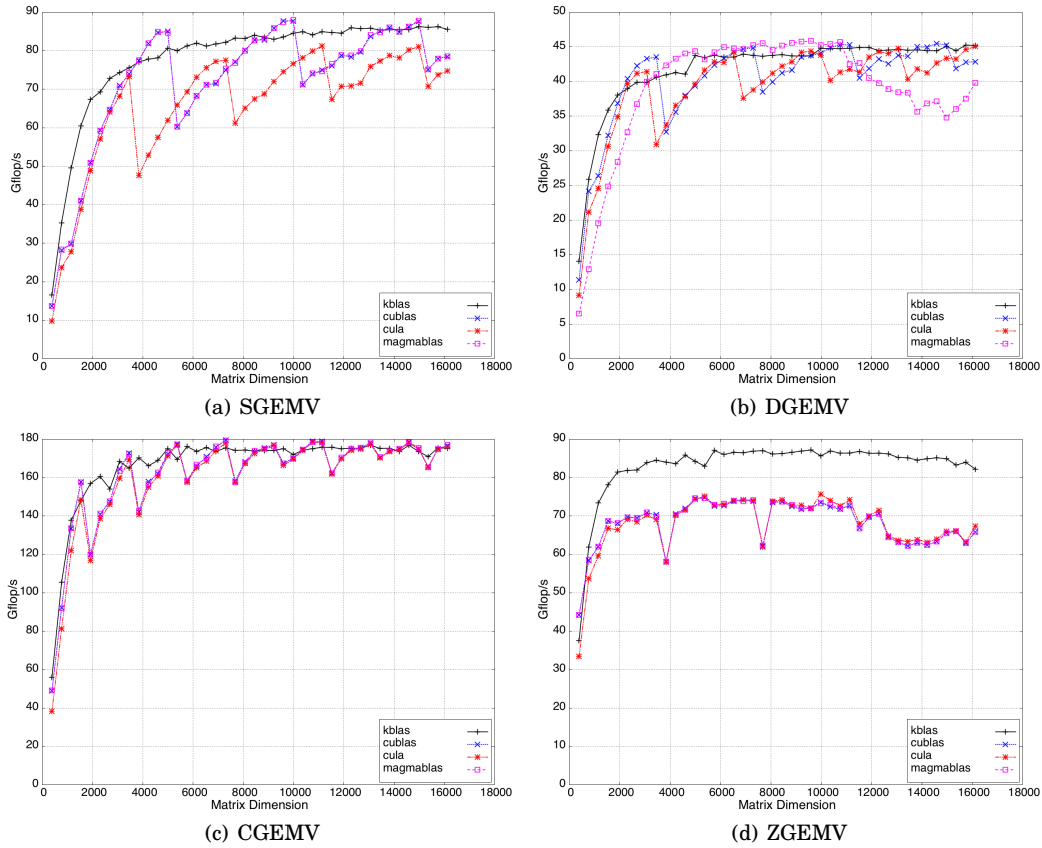


Fig. 9: GEMV Performance on a K20c GPU, ECC off

**8.2.2. Performance of SYMV/HEMV.** When the matrix is Hermitian, KBLAS scores speedups against all other implementations. Against the best competitor (MAGMABLAS), KBLAS asymptotic speedup is up to 50%, 15%, 24%, 55% across all precisions, as shown in Figure 10. Comparing against the bounds listed in Table IV, the performance is up to 89%, 87%, 90%, and 72% of the sustained peak performance. Looking at relatively small matrices (4k or less), the speedups against MAGMABLAS are up to 2.46x, 1.83x, 1.80x, and 1.89x across all precisions. A variant of the KBLAS SYMV/HEMV kernel has been integrated into NVIDIA's cuBLAS library, starting version 6.0.

**8.2.3. Performance for Submatrix Multiplication.** Figures 11 and 12 show the performance of the GEMV and the SYMV/HEMV kernels when the multiplication is done on a submatrix. The tests represented by these figures are done on submatrices that are part of a larger  $16384 \times 16384$  matrix. These submatrices are obtained by skipping certain number of rows and columns from the original matrix. We show the performance of the standard KBLAS kernels as well as the new-interface kernels that can compensate the memory non-coalesced access, as mentioned in Section 6.6. Figure 11 shows that all implementations suffer from the effect of the non-coalesced memory access. Only the KBLAS GEMV-OFFSET kernel manages to maintain the same high perfor-

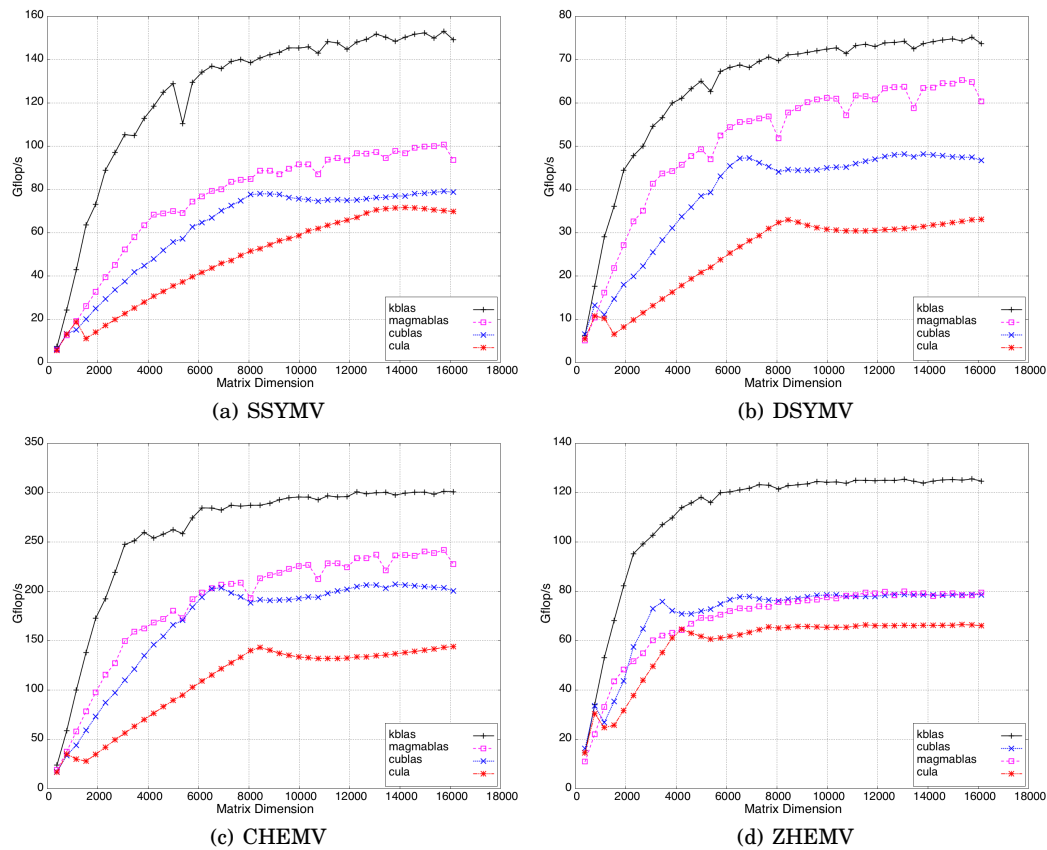


Fig. 10: SYMV/HEMV Performance on a K20c GPU, ECC off

mance shown in Figure 9. A similar behavior is observed for the SYMV/HEMV kernel in Figure 12.

An interesting observation is the performance spikes for standard implementation that arise for certain dimensions. These spikes happens when the offsets in rows and columns accidentally result in a submatrix that can read in fully coalesced manner. For example, consider Figure 12(a), where performance spikes arise at sizes like 5184, 9184, 13184, and others. These dimensions represent offsets that are multiples of 32, which represent multiples of 128 bytes in single precision. In general, if the offset, expressed in bytes, is multiple of 128, then the performance of the standard KBLAS implementation does not encounter any drops. We also observe minor spikes when the offset modulo 32 is either close to 0 or 32. In other precisions, the same analysis applies, but 32 should be replaced by 16 for double and single complex precisions, and with 8 for the double complex precision.

### 8.3. Multi-GPU Performance

Figures 13 and 14 show the performance of the KBLAS GEMV and SYMV/HEMV kernels on multi-GPUs. Matrices are stored in the data layout shown in Figure 3. While a multi-GPU GEMV kernel can be factored into calls to the single GPU GEMV kernel, a multi-GPU SYMV/HEMV cannot be factored in the same way. This is because the rectangular submatrices lose symmetry, and so a sophisticated kernel is required.

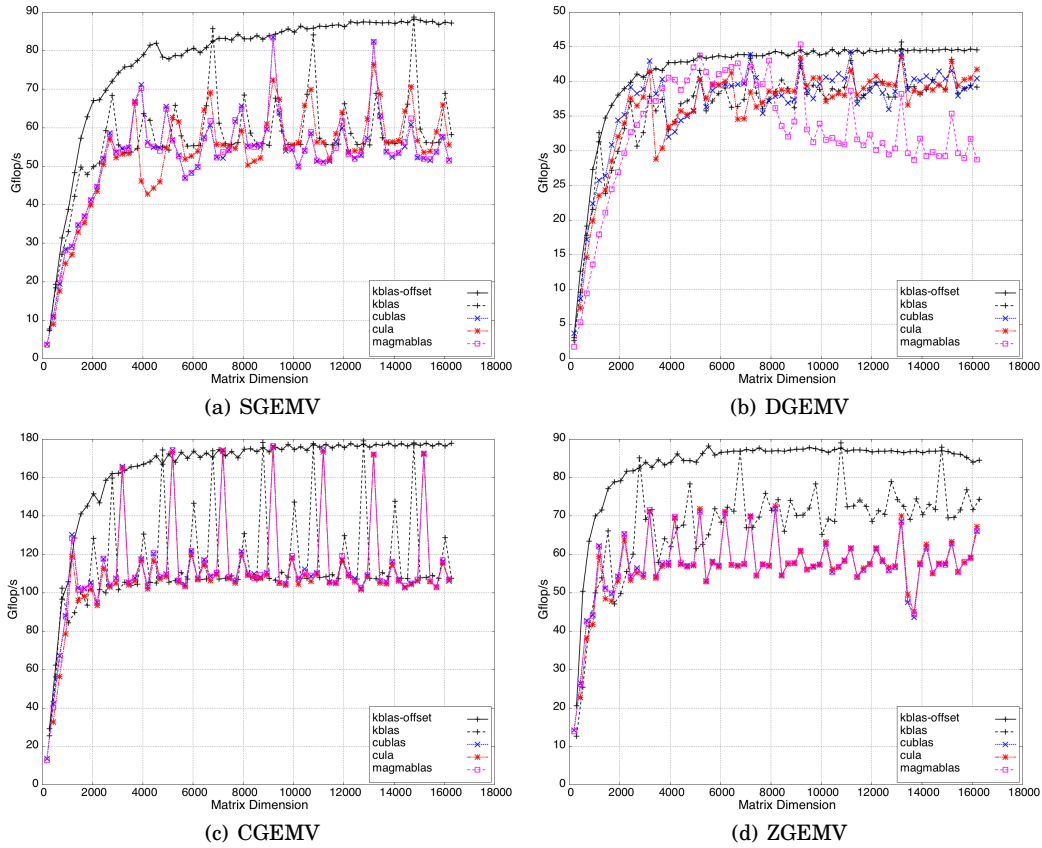


Fig. 11: Performance of GEMV by a submatrix on a K20c GPU, ECC off

KBLAS, however, still provides a sophisticated GEMV kernel on multi-GPU. This kernel takes into account the multiplication by a submatrix in the same manner described in Section 6.6. It is expected, therefore, to perform better than calling standard KBLAS-GEMV kernels on the local submatrices. The KBLAS GEMV-MGPU kernel is very close to strong scaling on up to 8 GPUs on a single node. This kernel can be used in matrix reduction techniques on large non-symmetric matrices that do not fit in single GPU memory. Considering the case when the matrix is Hermitian, shown in Figure 14, the KBLAS SYMV/HEMV kernel on multi-GPUs can achieve up to 43%, 38%, 38%, and 61% performance improvement against MAGMABLAS on 8 GPUs for all four precisions. According to the authors' knowledge, only MAGMABLAS and KBLAS provide a multi-GPU SYMV/HEMV kernel.

## 9. PERFORMANCE TUNING

The performance of KBLAS is controllable through a set of tuning parameters. Such parameters should be tuned according to the GPU architecture/model as well as the CUDA runtime version. For best performance, KBLAS should be retuned whenever the GPU changes or the CUDA runtime is upgraded.

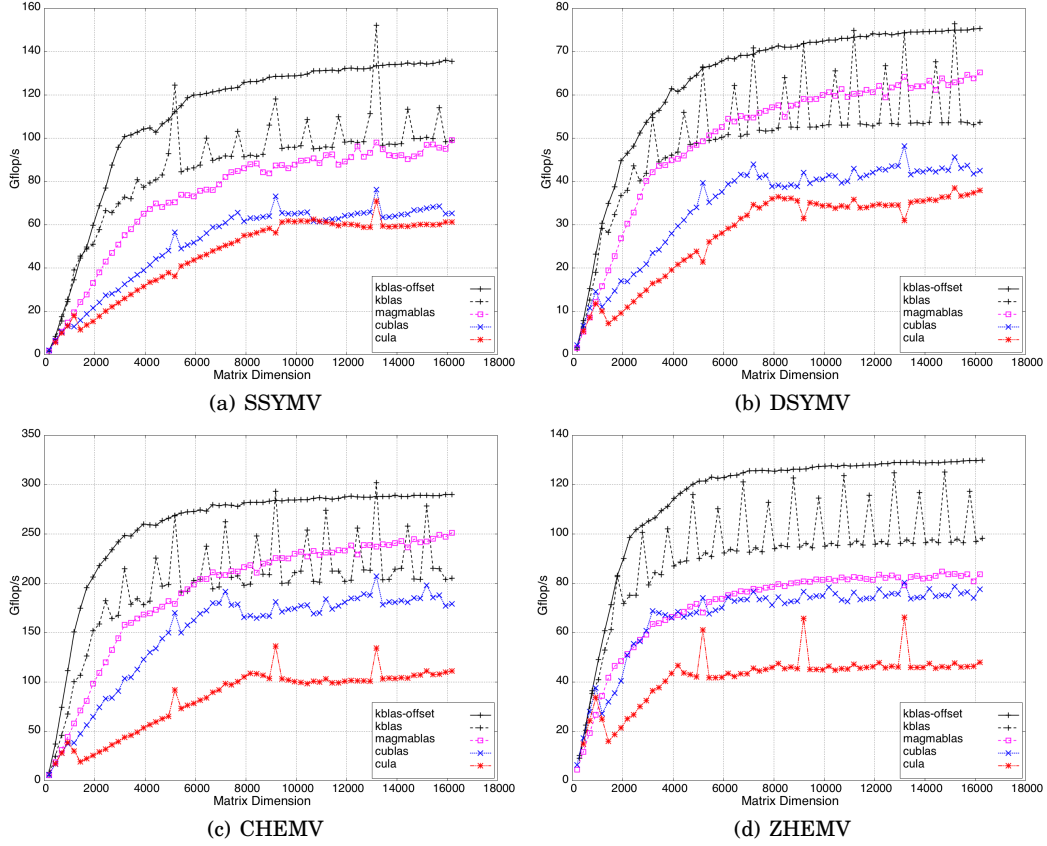


Fig. 12: Performance of SYMV/HEMV by a submatrix on a K20c GPU, ECC off

### 9.1. Tuning Process of a KBLAS Kernel

Any KBLAS kernel has at most three tuning parameters: the block size  $nb$ , the number of thread columns in a TB  $\bar{Q}$  (each of which has  $nb$  threads), and the number of TBs that collaboratively process a block row or a block column of the input matrix ( $\bar{Y}$ ). Both  $nb$  and  $\bar{Q}$  are used for *coarse tuning*, while  $\bar{Y}$  is used for *fine tuning*. It is advised, therefore, to tune first  $nb$  and  $\bar{Q}$ . We will discuss a case study of tuning the DSYMV kernel on a K20c GPU. All KBLAS kernels are tunable in the same way mentioned below.

At the stage of *coarse tuning*, the parameter  $\bar{Y}$  should be set to 1. Our experiments show that both values of  $nb$  and  $\bar{Q}$  should be powers of two, for best performance. Additionally,  $nb$  is multiples of the warp size in most cases. The value range of  $\bar{Q}$  is restricted by Equation 7, since  $L$  should be an integer greater than zero. This means that the parameter space for the  $(nb, \bar{Q})$  is relatively small, and that hand tuning of KBLAS can be done in a reasonable amount of time.

Figure 15(a) shows the performance of the DSYMV kernel for different values of the  $(nb, \bar{Q})$ , with  $\bar{Y}$  fixed at 1. At such stage of tuning, the focus should be on the asymptotic performance, since the behavior for relatively small matrices is usually impacted by  $\bar{Y}$ . Figure 15(a) shows that the best asymptotic performance is achieved by the (32, 2, 1) configuration.

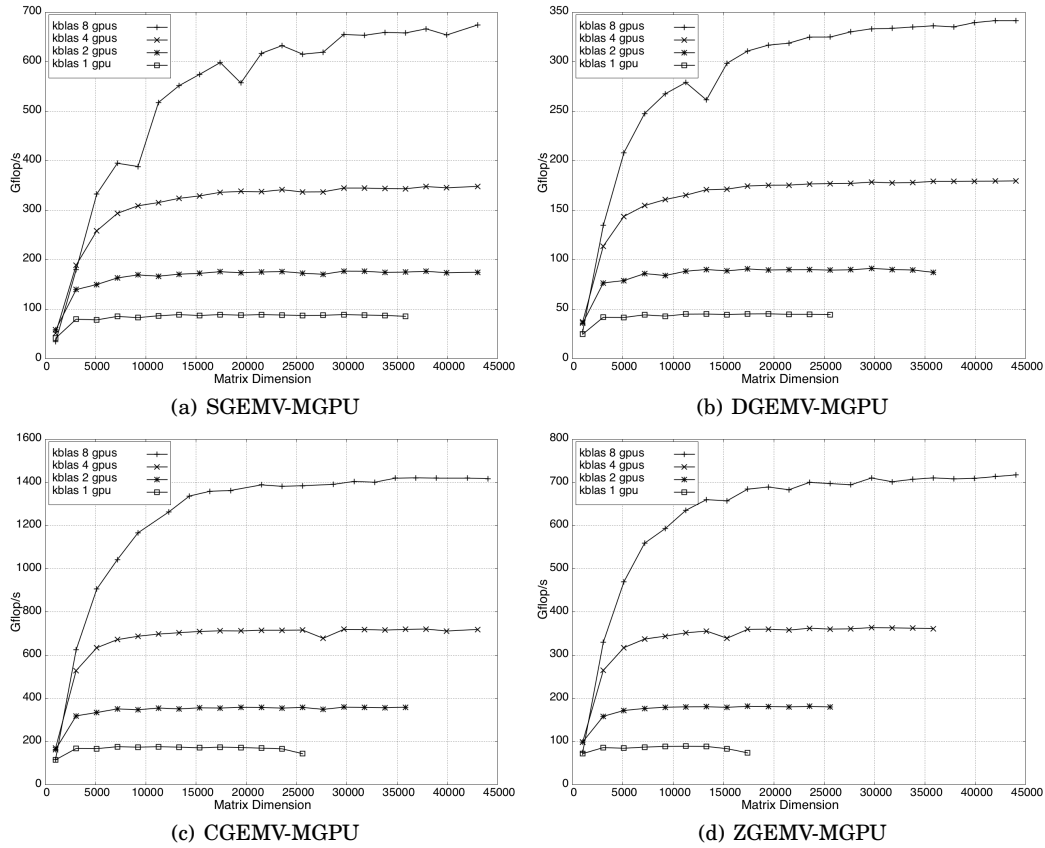


Fig. 13: GEMV Performance on Multi-GPU, K20c with ECC off

The next stage is to try increasing  $\bar{Y}$  while fixing  $(nb, \bar{Q})$ . Figure 15 shows that the best performance is the (32, 2, 2) configuration. As expected, increasing the value of  $\bar{Y}$  enhances the performance for relatively small matrices. But on the other hand, a too large value of  $\bar{Y}$  (like the (32, 2, 16) configuration) means more pressure on atomic operations, and therefore, the performance becomes negatively influenced.

The simple strategy mentioned above works well on several GPU architectures/models. Figure 16 shows the performance of the DSYMV on different GPUs: A Fermi M2090, a Kepler K20c, a Kepler K40c, and a GTX Titan. The respective sustained peak memory bandwidths of these GPUs, as scored by the STREAM benchmark, are 130.32 GB/s, 175.24 GB/s, 219.65 GB/s, and 239.87 GB/s, respectively. All performance curves in this figure have been tuned in the same way. Leveraging the same performance model mentioned in Section 7 on these GPUs, the asymptotic performance of the DSYMV kernel is up to 77%, 87%, 78%, and 84% on the aforementioned GPUs.

## 9.2. Smoothing GEMV Performance

Another interesting point is the impact of the  $\bar{Y}$  value on the performance of the GEMV kernel. Consider the DGEMV performance in Figure 17 on a K20c GPU, which has 13 SMs. We observe an oscillatory behavior in the performance of the (64, 8, 1) configuration. Such configuration is typical for the previous GEMV kernel proposed by the

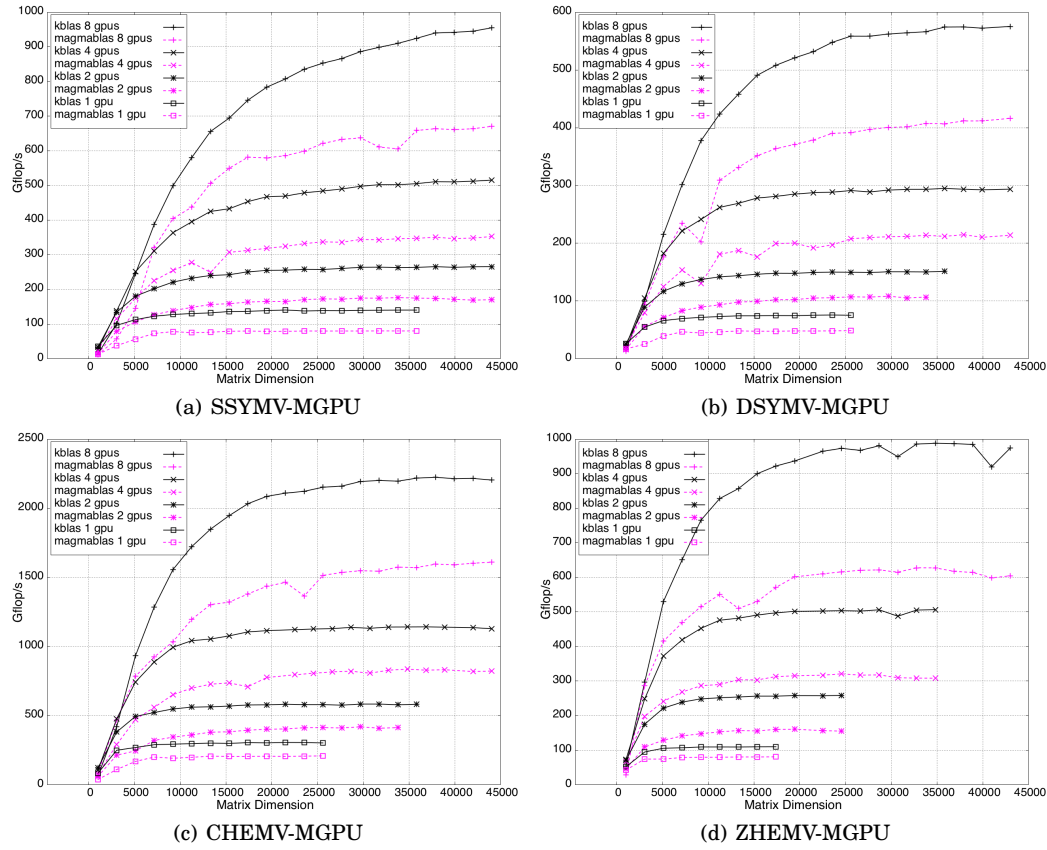


Fig. 14: SYMV/HEMV Performance on Multi-GPU, K20c with ECC off

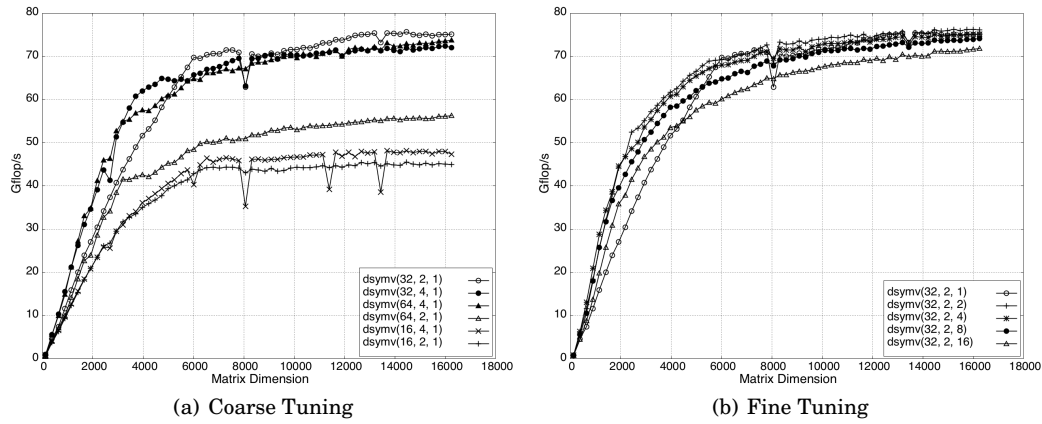


Fig. 15: Performance Tuning of KBLAS-DSYMV Kernel on a K20c GPU

authors [Abdelfattah et al. 2013a] in the sense that it does not expose  $\bar{Y}$  as a tuning parameter and keeps it fixed at 1. The oscillatory behavior is already discussed in Section



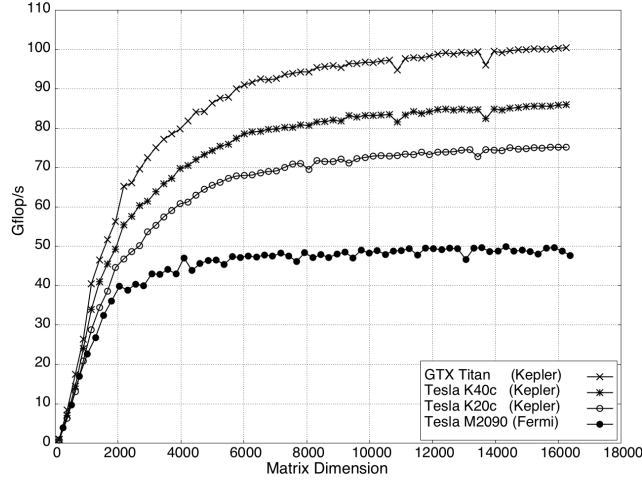


Fig. 16: Performance of a Tuned KBLAS-DSYMV Kernel on Different GPUs

4.3.4. The GEMV kernel launches TBs with balanced workloads, and so it encounters oscillations in performance when the input matrix dimension leads to a low  $TB_R$ . The performance drops in the (64, 8, 1) configuration come for dimensions that have low  $TB_R$  values. For example, dimensions like 1792, 3456, 5120, 6784, 8448, and 10112 have huge drops in performance. These dimensions all end up with  $TB_R=2$ . The drops are periodic but get smaller, as the number of full utilization rounds gets larger and the GPU becomes closer to a full utilization during most of the kernel execution time.

The ability of KBLAS to incorporate more than a TB per block row can compensate these oscillations. Since the execution time of a single TB becomes smaller when  $\bar{Y}$  increases, the number of rounds of full utilization is significantly increased, while the round of partial utilization is always fixed at 1. The time spent by the GPU in the partial utilization round with respect to the time spent in full rounds becomes smaller and even negligible.

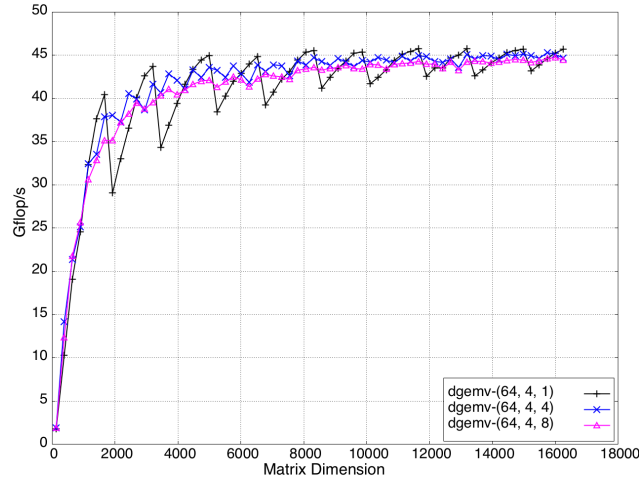


Fig. 17: Impact of  $\bar{Y}$  on the Performance of the KBLAS-DGEMV Kernel

Figure 17 shows the impact of increasing  $\bar{Y}$ . The performance drops gets smaller as  $\bar{Y}$  increases. However, the asymptotic performance slightly drops due to the overhead of atomic operations. Recall that the GEMV kernel does exactly  $\bar{Y}$  atomic additions of size  $nb$  per a block row or a block column of the input matrix.

## 10. ACCELERATING EXISTING NUMERICAL LINEAR ALGEBRA LIBRARIES

In this section, we show the performance improvement that KBLAS can achieve when it is integrated into higher level algorithms that extensively use dense matrix vector multiplication. This case study focuses on LAPACK algorithms provided by MAGMA. The system setup is the same as described in Section 8.1.

MAGMA is an open source library that provides optimized BLAS and LAPACK routines for multi-core architecture accelerated using GPUs [MAGMA 2009] [Agullo et al. 2009]. We pick two algorithms from MAGMA and accelerate them using KBLAS: The bidiagonal reduction algorithm for general matrices (GEBRD); and the tridiagonal reduction for symmetric/Hermitian matrices (SYTRD/HETRD). These two algorithms are used in Singular Value Decomposition (SVD), and Eigenvalue Decomposition (EVD) for dense matrices, respectively. They are implemented using block algorithms, where the compute intensive components are offloaded to the GPUs, as proposed in [Tomov et al. 2010]. Since KBLAS provides two implementations for each of the GEMV and the SYMV/HEMV kernels, we will show the impact on the performance of the aforementioned algorithms with each implementation.

Figure 18 shows the performance of the MAGMA GEBRD algorithm for all four precisions on a single GPU. The original MAGMA implementation uses cuBLAS GEMV kernel to update the unreduced part of the input matrix. As pointed in Section 8.2, the performance of KBLAS GEMV is very similar to its cuBLAS counterpart as shown in Figure 9. Therefore, the performance of the GEBRD algorithm is either very similar or slightly better than the original MAGMA implementation. However, the impact of using the KBLAS GEMV-OFFSET kernel is more significant than the KBLAS GEMV. This is due to the fact that the former achieves a much better performance if the multiplication is done by a submatrix, as we showed in Figure 11. The improvements in the GEBRD performance are up to 31%, 29%, 61%, and 71% on all four precisions. Given that the KBLAS GEMV-OFFSET kernel does extra reads from global memory, its significance to the GEBRD appears only for relatively large matrices, where the extra reads become negligible.

Figure 19 shows the performance improvement of the MAGMA SYTRD/HETRD algorithm when the KBLAS SYMV/HEMV kernel (both implementations) is incorporated instead of the original implementation by MAGMABLAS. Similar to the GEBRD algorithm, the new-interface kernels from KBLAS are used to give the best possible performance. The improvements are up to 35%, 59%, 56%, and 49% for all precisions. We notice that the best performance gain comes for relatively small matrices, because KBLAS significantly improves the performance of the DSYMV/HEMV kernel for these sizes of matrices against MAGMABLAS, as shown in Figure 12.

While MAGMA does not provide a multi-GPU bidiagonal reduction (in which KBLAS GEMV-MGPU kernel can be used), it provides an implementation of the SYTRD/HETRD algorithm on multi-GPUs [Yamazaki et al. 2013]. Figure 20 shows the performance of this implementation when KBLAS is used instead of MAGMABLAS. Using 8 GPUs, the performance gains are up to 140%, 103%, 105%, and 62% for all four precisions. We notice that the original MAGMA implementation requires an initialization step for the memory workspace each time before calling the SYMV/HEMV kernel. KBLAS, in addition to its better performance, saves such initialization time, since it does not need any extra workspace. So, the overall performance

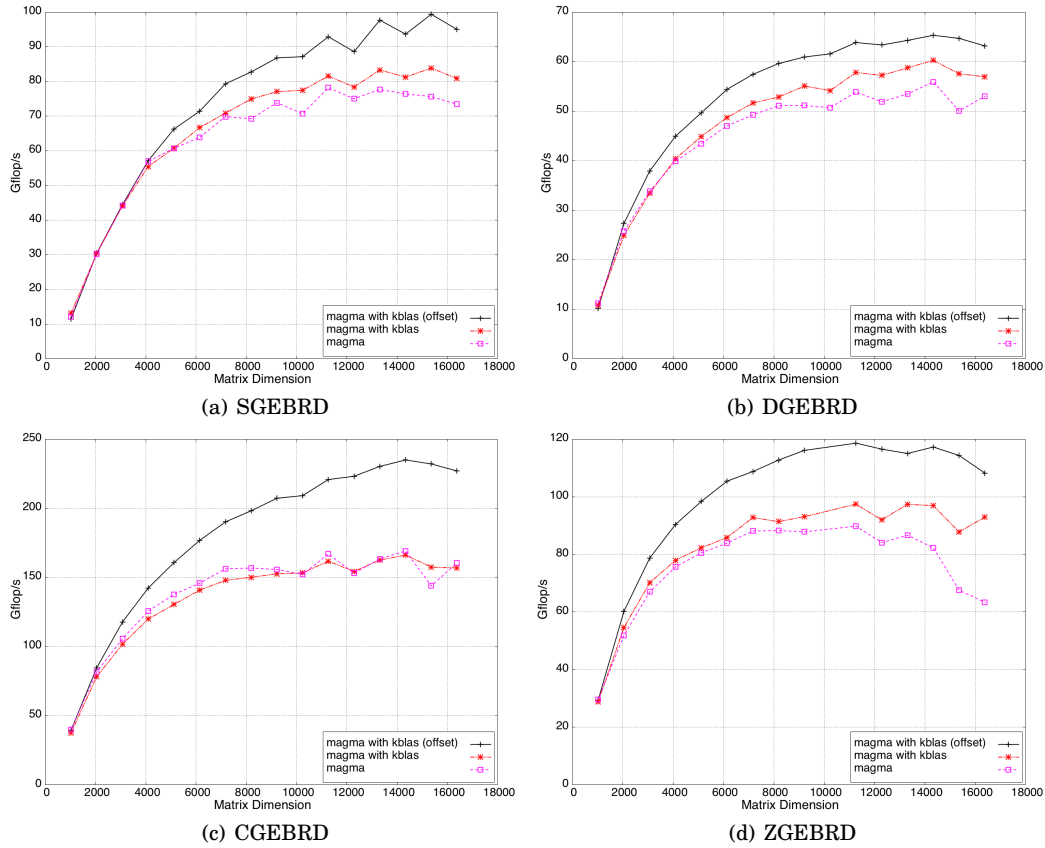


Fig. 18: Bidiagonal Reduction Performance on a K20c GPU, ECC off

gain comes from more optimized kernel + the saving in initialization time. The KBLAS-DSYMV kernel on multi-GPU has been used in accelerating a dense symmetric eigen solver for very large matrices in a computational astronomy application [Abdelfattah et al. 2014].

## 11. CONCLUSION AND FUTURE WORK

This paper introduces KBLAS, an open source library that provides optimized kernels for dense matrix vector multiplication using NVIDIA GPUs. Through a set of low-level optimizations, KBLAS outperform state-of-the-art implementations. Our experiments show that the performance is portable across different GPU models and architectures, thanks to the tuning parameters KBLAS provides for each kernel. The paper also shows that KBLAS can accelerate existing implementations of LAPACK algorithms.

In the future, KBLAS will continue to provide optimized BLAS kernels, where a room for improvement is envisioned. In addition, more data layouts for multi-GPU routines are to be supported, for example like the 2D cyclic format used in ScaLAPACK. This step is intended to support BLAS operations on distributed systems with multi-GPU accelerated nodes. Another direction is to provide a sophisticated auto-tuning functionality within KBLAS to facilitate kernel tuning on any GPU. This will also open the door for inserting more tuning parameters that can give the user more fine grain control on performance. We also plan to use KBLAS building blocks in Sparse

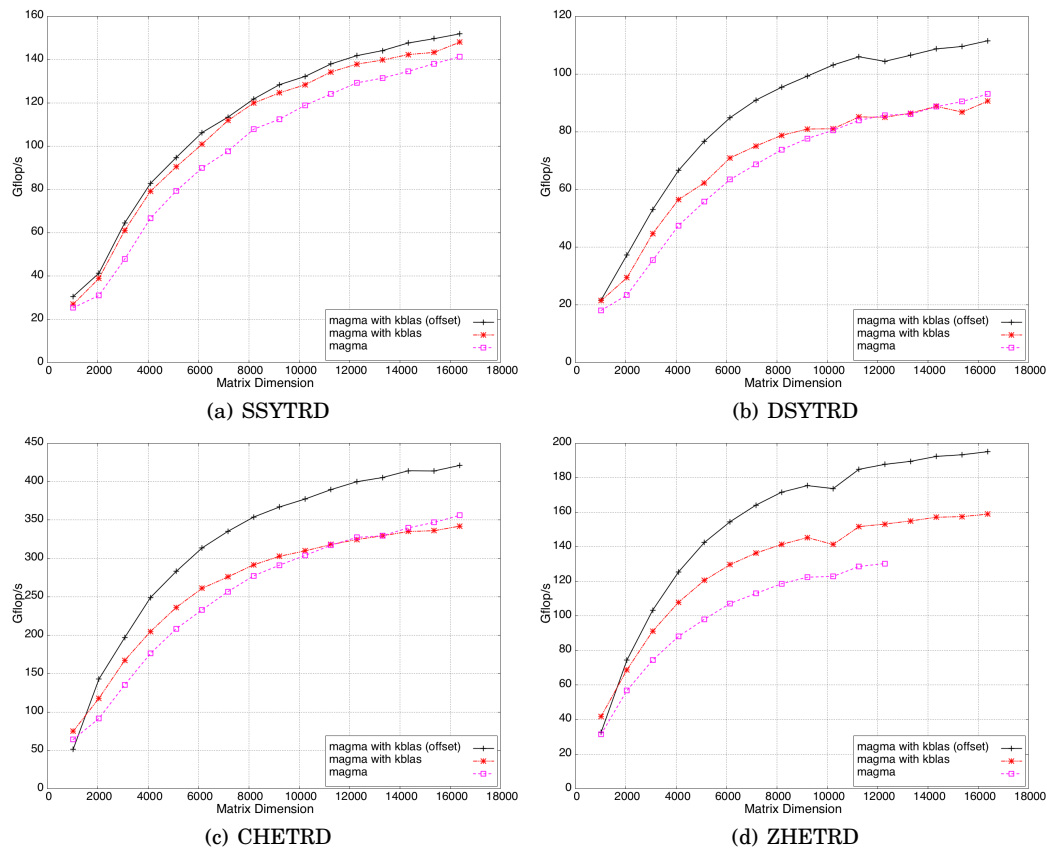


Fig. 19: Tridiagonal Reduction Performance on a K20c GPU, ECC off

Matrix Vector Multiplication (SpMV) for sparse matrices that have a substructure of dense blocks.

### Acknowledgment

We would like to thank NVIDIA for their hardware donations and in particular, we would like to give credits to both Philippe Vandermersch and Sharan Chetlur from NVIDIA for their help and support during the integration of the SYMV/HEMV routines into cuBLAS library. We also thank the CSCS Swiss National Supercomputing Centre for providing access to their GPU computing platforms.

### REFERENCES

- ABDELFATTAH, A., DONGARRA, J., KEYES, D., AND LTAIEF, H. 2013a. Optimizing Memory-Bound SYMV Kernel on GPU Hardware Accelerators. In *High Performance Computing for Computational Science - VECPAR 2012*, M. Dayd, O. Marques, and K. Nakajima, Eds. Lecture Notes in Computer Science Series, vol. 7851. Springer Berlin Heidelberg, 72–79.
- ABDELFATTAH, A., GENDRON, E., GRATADOUR, D., KEYES, D., LTAIEF, H., SEVIN, A., AND VIDAL, F. 2014. High Performance Pseudo-analytical Simulation of Multi-Object Adaptive Optics over Multi-GPU Systems. In *Euro-Par 2014 Parallel Processing*, F. Silva, I. Dutra, and V. Santos Costa, Eds. Lecture Notes in Computer Science Series, vol. 8632. Springer International Publishing, 704–715.
- ABDELFATTAH, A., KEYES, D., AND LTAIEF, H. 2013b. Systematic Approach in Optimizing Numerical Memory-Bound Kernels on GPU. In *Euro-Par 2012: Parallel Processing Workshops*, I. Caragiannis,

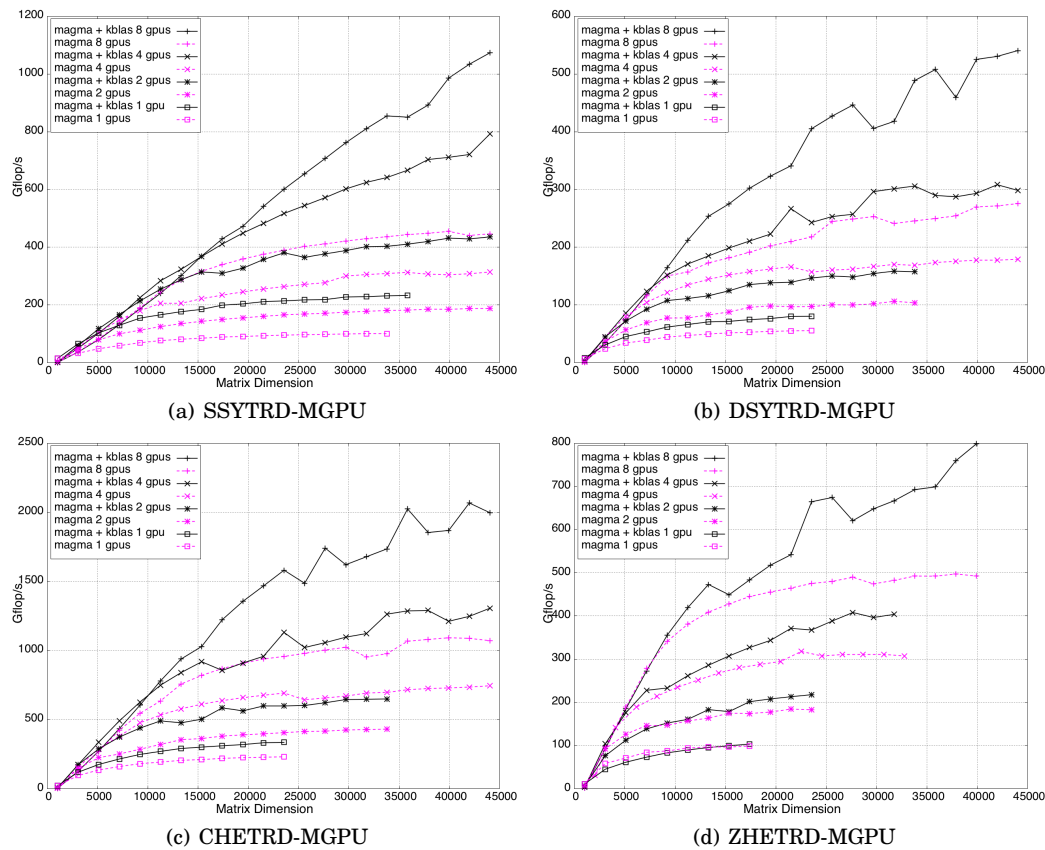


Fig. 20: Tridiagonal Reduction Performance on Multi-GPUs, K20c GPU with ECC off

- M. Alexander, R. Badia, M. Cannataro, A. Costan, M. Danelutto, F. Desprez, B. Krammer, J. Sahuquillo, S. Scott, and J. Weidendorfer, Eds. Lecture Notes in Computer Science Series, vol. 7640. Springer Berlin Heidelberg, 207–216.
- AGULLO, E., DEMMEL, J., DONGARRA, J., HADRI, B., KURZAK, J., LANGOU, J., LTAIEF, H., LUSZCZEK, P., AND TOMOV, S. 2009. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series* 180.
- ANDERSON, E., BAI, Z., BISCHOF, C., BLACKFORD, S. L., DEMMEL, J. W., DONGARRA, J. J., CROZ, J. D., GREENBAUM, A., HAMMARLING, S., MCKENNEY, A., AND SORESENSEN, D. C. 1999. *LAPACK User's Guide* 3rd Ed. Society for Industrial and Applied Mathematics, Philadelphia.
- BLAS. Basic Linear Algebra Subprograms. <http://www.netlib.org/blas/>.
- BUCK, I., FOLEY, T., HORN, D., SUGERMAN, J., FATAHALIAN, K., HOUSTON, M., AND HANRAHAN, P. 2004. Brook for GPUs: Stream Computing on Graphics Hardware. In *ACM SIGGRAPH 2004 Papers*. SIGGRAPH '04. ACM, New York, NY, USA, 777–786.
- HUMPHREY, J., PRICE, D., SPAGNOLI, K., PAOLINI, A., AND KERMELIS, E. 2010. CULA: Hybrid GPU Accelerated Linear Algebra Routines. In *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*. Vol. 7705. 1.
- KIRK, D. B. AND HWU, W.-M. W. 2010. *Programming Massively Parallel Processors: A Hands-on Approach* 1st Ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
- MAGMA. 2009. Matrix Algebra on GPU and Multicore Architectures. Innovative Computing Laboratory, University of Tennessee. Available at <http://icl.cs.utk.edu/magma/>.

- MCCALPIN, J. D. 1991-2007. STREAM: Sustainable Memory Bandwidth in High Performance Computers. Tech. rep., University of Virginia, Charlottesville, Virginia. A continually updated technical report. <http://www.cs.virginia.edu/stream/>.
- MCCALPIN, J. D. 1995. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, 19–25.
- NATH, R., TOMOV, S., DONG, T. T., AND DONGARRA, J. 2011. Optimizing Symmetric Dense Matrix-vector Multiplication on GPUs. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '11. ACM, New York, NY, USA, 6:1–6:10.
- NATH, R., TOMOV, S., AND DONGARRA, J. 2010a. An Improved Magma Gemm For Fermi Graphics Processing Units. *Int. J. High Perform. Comput. Appl.* 24, 4, 511–515.
- NATH, R., TOMOV, S., AND DONGARRA, J. 2010b. *BLAS for GPUs*. CRC Press, 57–80.
- NATH, R., TOMOV, S., AND DONGARRA, J. 2011. Accelerating GPU Kernels for Dense Linear Algebra. In *Proceedings of the 9th International Conference on High Performance Computing for Computational Science*. VECPAR'10. Springer-Verlag, Berlin, Heidelberg, 83–92.
- NVIDIA. 2009. NVIDIA Fermi Compute Architecture Whitepaper. [http://www.nvidia.com/content/PDF/fermi\\_white\\_papers/NVIDIA\\_Fermi\\_Compute\\_Architecture\\_Whitepaper.pdf](http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf).
- NVIDIA. 2012. NVIDIA Kepler GK110 Architecture Whitepaper. <http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf>.
- NVIDIA. 2014a. CUDA C Programming Guide. <http://docs.nvidia.com/cuda/cuda-c-programming-guide/>.
- NVIDIA. 2014b. The NVIDIA CUDA Basic Linear Algebra Subroutines. <https://developer.nvidia.com/cublas/>.
- TAN, G., LI, L., TRIECHLE, S., PHILLIPS, E., BAO, Y., AND SUN, N. 2011. Fast Implementation of DGEMM on Fermi GPU. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '11. ACM, New York, NY, USA, 35:1–35:11.
- TOMOV, S., NATH, R., AND DONGARRA, J. 2010. Accelerating the Reduction to Upper Hessenberg, Tridiagonal, and Bidiagonal Forms Through Hybrid GPU-based Computing. *Parallel Comput.* 36, 12, 645–654.
- VOLKOV, V. AND DEMMEL, J. 2008. Benchmarking GPUs to Tune Dense Linear Algebra. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*. 1–11.
- WILLIAMS, S., WATERMAN, A., AND PATTERSON, D. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* 52, 4, 65–76.
- YAMAZAKI, I., DONG, T., SOLC, R., TOMOV, S., DONGARRA, J., AND SCHULTHESS, T. 2013. Tridiagonalization of a Dense Symmetric Matrix on Multiple GPUs and its Application to Symmetric Eigenvalue Problems. *Concurrency and Computation: Practice and Experience*.